Control and Cybernetics

vol. **39** (2010) No. 4

Fail-bounded implementations of the numerical model predictive control algorithms*

by

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

[1] Institute of Computer Science, Warsaw University of Technology
ul. Nowowiejska 15/19,00-665 Warszawa, Poland

[2]Institute of Control and Computation Engineering,
Warsaw University of Technology
ul. Nowowiejska 15/19, 00-665 Warszawa, Poland

e-mail: {P.Gawkowski, J.Sosnowski}@ii.pw.edu.pl,
{M.Lawrynczuk, P.Marusak, P.Tatjewski}@ia.pw.edu.pl

Abstract: Methods of fault-hardening software implementations of the numerical Model Predictive Control (MPC) algorithms are discussed in the paper. In particular, Generalized Predictive Control (GPC) algorithms are considered. The robustness of these algorithms with respect to faults is crucial for process safety and economic efficiency, as faults may result in major control performance degradation or even destabilization. Therefore, fault-hardening of GPC algorithms is an important issue. The fault sensitivity of the non-fault-hardened algorithms implementations and the effectiveness of the fault hardening procedures are verified in experiments with a software implemented fault injector. These experiments refer to the control system of a chemical plant. Experience with fault simulations resulted in some methods of fault-hardening which are described in detail. Improvement of the dependability of the GPC algorithms is commented for each of the proposed fault-hardening mechanism.

Keywords: dependability, fault simulation, robust process control, GPC.

1. Introduction

High dependability and, in particular, safety are important features required in many control systems used in industry, automotive systems, medicine, civil engineering applications, etc. The respective control systems usually exploit

*Submitted: January 2010; Accepted: July 2010.

feedback, where signals from a physical process are compared against a set of reference values and the controller produces output signals that maintain the controlled process within a required state space. Output signals should be correct and delivered on time to assure the specified goal (e.g. safe plane landing, reliable stopping of a car). The control process can be disturbed by various internal or external faults. An important issue is to analyse system susceptibility to such faults as well as to alleviate fault effects by detecting, masking or tolerating them, to avoid unsafe situations. For this purpose various fault simulation techniques have been proposed and described in the literature, see, e.g., Anghel, Leveugle and Vanhauwaert (2005) or Arlat et al. (2003). Most of them were used to study the calculation oriented applications, so fault effects were qualified as correct, incorrect and detected (e.g. by generating a system exception) by analyzing the final result for a specified input data. In the case of real time systems with control feedback, the fault effect analysis is more complex, because we have to trace the generated control signals in time, take into account the reaction of the controlled object and qualify its behaviour. This process is application dependent. In the literature such studies are rarely encountered and in most cases they relate to simple control algorithms (e.g. based on PID controller) and dedicated simulation platforms, see, e.g., Corno et al. (2004), Gaid et al. (2006), Mariani, Fuhrmann and Vittorelli (2006), Nouillant et al. (2002). In this paper we present a more universal approach, based on software implemented fault injector (SWIFI), which disturbs the execution of the control algorithm in real computer environment. It is adapted to allow the analysis of reactive systems and equipped with the system behaviour analyser to qualify fault effects.

Our fault injection tool (FITS, see Sosnowski, Gawkowski and Lesiak, 2004), developed according to the SWIFI concept, allowed us to study sophisticated control algorithms and improve their resistivity to faults. In particular, it traces real fault effects, which is important for process control designers who can introduce effective fault hardening mechanisms in software. The most effective approaches are highly redundant systems based on hardware replication and voting, which are used in some critical applications such as avionics, nuclear power plants, life supporting systems, etc. (Baleani et al., 2003; Gaid et al., 2006; Gawkowski and Sosnowski, 2001, 2003, 2005; Hecht, 1979; Isermann, 2006; Korbicz et al., 2004). In many applications, especially process control, such redundancy is not economically acceptable; moreover, for longer response times and specific control object properties (e.g. high inertia, natural fault tolerance by auto correction in the feedback loop) some simpler solutions are possible. Additionally, in these applications we can concentrate on transient and intermittent faults which dominate (typically their frequency is 100-1000 times higher than for permanent faults), Anghel, Leveugle and Vanhauwaert (2005), Gawkowski and Sosnowski (2001).

In contrast to other approaches, we distinguish latched (e.g. bit flip in RAM cell) and non-latched (e.g. bus line disturbance) transient faults to deal with

RAM and flash memory based controllers. In the literature three failure models are discussed, Cunha, Rela and Silva (2002):

– fail-arbitrary: controller produces any outputs (including not correct and unsafe),
– fail-silent: controller produces either correct outputs or no outputs, but in the latter case the physical application is put into a safe state,
– fail-bounded: controller produces correct outputs, no outputs after detecting some errors (to assure a safe state) and wrong outputs within specified boundaries in time and value (for a specified class of errors).

Fail-arbitrary model relates to systems with no fault detection or handling mechanisms. Fail-silent model can be assured by controller duplication and a comparator, Baleani et al. (2003), Isermann (2006). In practice, all controllers based on COTS microprocessors comprise some fault detection mechanisms (e.g. exceptions, parity checkers), so they are adequate for fail-bounded model. Moreover, fault handling can be improved at the software level, Gawkowski and Sosnowski (2003, 2005), Hecht (1979), Skarin and Karlsson (2008). Hence, we concentrate on the fail-bounded model. However, we have enhanced this model by defining quality measures of the performed tasks, which would allow for admitting higher rate of temporary erroneous output signals.

Behaviour of the system can be evaluated by various assertions within the performed control algorithm or directly on the controlled object. In the first approach we can use a simpler algorithm in parallel with the numerical algorithm to evaluate the progress of the control variables (internal or output signals). In the second approach, instead of tracing the series of output signals from the controller, we analyse the behaviour of the controlled object itself by tracing its state space within the acceptable execution and stability region (band) under the assumed service quality level. For this purpose, some knowledge of the control problem is needed. In the literature, the authors considered mostly controlled objects with a uniquely specified safety state, e.g. in inverse pendulum – the state of not falling down, Cunha, Rela and Silva (2002), in automotive controllers situations resulting into running off the road e.g. due to a fault in the ABS or car suspension system, Gaid et al. (2006), Skarin and Karlsson (2008), Trawczyński, Sosnowski and Gawkowski (2008). This leads to defining some critical values of the controlled parameters. Many experiments were developed for the simplified models and only for a small number of specific faults (e.g. disturbing some control variables). In many industrial applications we deal with long term continuous processes with high inertia and delays, so the evaluation of the correctness of the controlling process is distributed in time. We have found that Sum of Squared Errors (SSE) of the controlled parameters is a good choice for performing this evaluation. Such complex evaluation is provided by our tool FITS.

Section 2 presents the basic concept of model predictive control (MPC) algorithms. In particular, the explicit and numerical GPC algorithms are described.

The experimental evaluation methodology is presented in Section 3. Various fault-hardening methods are described in Section 4 and their effectiveness is illustrated with experimental results. The paper concludes with the last section.

2. Generalized predictive control algorithms

In the MPC algorithms, Ławryńczuk, Marusak and Tatjewski (2008), Maciejowski (2002), Rossiter (2003), Tatjewski (2007), a dynamic model of the process is used in order to predict its future behaviour and to calculate the optimal control policy (Fig. 1).

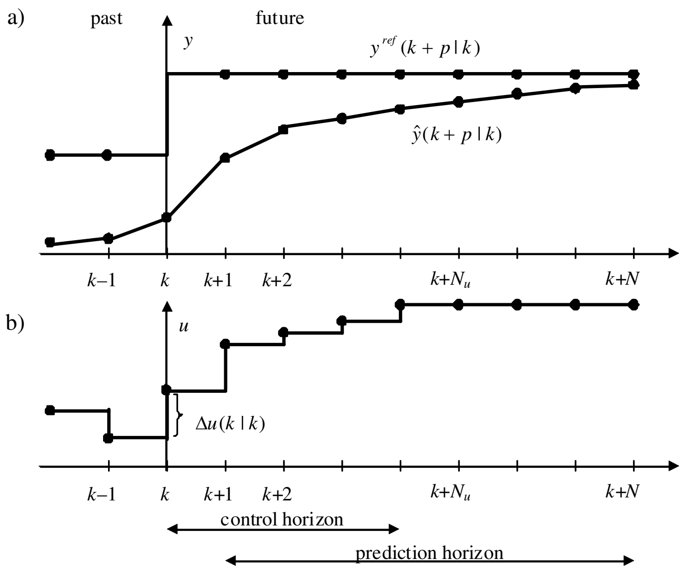

Figure 1. The principle of MPC algorithms; a) the predicted output signal, b) the control signal

More specifically, at each consecutive sampling instant k a set of future control increments $\Delta\boldsymbol{u}(k)$ is calculated, assuming that since the $k+N_u$-th sampling instant (where N_u is the *control horizon*) the values of control variables are constant (control increments are zero). The future control policy $\Delta\boldsymbol{u}(k)$ is calculated in such a way that future predicted control errors (i.e. the differences between the desired trajectory $\boldsymbol{y}^{ref}(k)$ and predicted values of the process output $\hat{\boldsymbol{y}}(k)$ – see Fig. 1) are minimised over the *prediction horizon* N, i.e. the following cost function is usually minimised:

$$J(k) = \left\|\boldsymbol{y}^{ref}(k) - \hat{\boldsymbol{y}}(k)\right\|^2_{\boldsymbol{M}} + \left\|\Delta\boldsymbol{u}(k)\right\|^2_{\boldsymbol{\Lambda}}, \tag{1}$$

where the reference trajectory $\boldsymbol{y}^{ref}(k)$ and the prediction of the process outputs $\hat{\boldsymbol{y}}(k)$ are vectors of length n_yN, whereas $\boldsymbol{M}$ and $\boldsymbol{\Lambda}$ are weighting matrices of dimensions $n_yN \times n_yN$ and $n_uN_u \times n_uN_u$, respectively; it is assumed that the process has n_u inputs and n_y outputs ($u \in \Re^{n_u}, y \in \Re^{n_y}$). Usually, the reference trajectory is assumed constant over the prediction horizon.

Generally, longer prediction horizon means that the controller modifies the present control (realised in the period defined by the control horizon) to better stabilize the control plant over the predicted N future sampling instants. Although in the GPC algorithm a number of future control increments over the control horizon are calculated, only the first n_u elements of the vector $\Delta\boldsymbol{u}(k)$ are actually applied to the process. At the next sampling instant $(k+1)$ the prediction is shifted one step forward and the whole procedure aiming at optimising the future control policy is repeated.

In the GPC algorithm, the predictions $\hat{y}(k+p|k)$ over the prediction horizon $(p = 1, ..., N)$ are calculated using the difference equation as the dynamic model of the process (details can be found in Camacho and Bordons, 1999; Maciejowski, 2002; Rossiter, 2003; Tatjewski, 2007).

Owing to the use of the linear model of the process, it is possible to express the output prediction as the sum of a forced trajectory, which depends only on future control increments $\Delta\boldsymbol{u}(k)$, and a free trajectory $\boldsymbol{y}^0(k)$, which depends only on the past:

$$\hat{\boldsymbol{y}}(k) = \boldsymbol{G}\Delta\boldsymbol{u}(k) + \boldsymbol{y}^0(k) \tag{2}$$

where the dynamic matrix $\boldsymbol{G}$ of dimensionality $n_yN \times n_uN_u$ consists of step-response coefficients of the process model.

2.1. The explicit GPC algorithm

When the cost function (1) is optimised without any constraints, the future vector of optimal control increments is calculated explicitly as

$$\Delta\boldsymbol{u}(k) = \boldsymbol{K}(\boldsymbol{y}^{ref}(k) - \boldsymbol{y}^0(k)) \tag{3}$$

where $\boldsymbol{K} = (\boldsymbol{G}^T\boldsymbol{M}\boldsymbol{G} + \boldsymbol{\Lambda})^{-1}\boldsymbol{G}^T\boldsymbol{M}$ is a matrix of dimensions $n_uN_u \times n_yN$, which can be calculated off-line, thus the inversion of the matrix during the operation is avoided as it is hard-coded. The control law (3) is a linear feedback from the set-point values, the values of the process outputs and the manipulated variable increments calculated at previous sampling instants. Although in the explicit GPC algorithm constraints are not taken into account during calculation of the values of manipulated variable, it is possible to apply projection of control signals on the constraint set, see, e.g., Tatjewski (2007).

2.2. The numerical GPC algorithm

In contrast to the explicit GPC algorithm, in the numerical GPC algorithm at each sampling instant the following quadratic programming problem with constraints must be solved in order to obtain optimal future control increments $\Delta\boldsymbol{u}(k)$

$$\min_{\Delta\boldsymbol{u}(k)} \{J(k) = \left\|\boldsymbol{y}^{ref}(k) - \hat{\boldsymbol{y}}(k)\right\|^2_{\boldsymbol{M}} + \|\Delta\boldsymbol{u}(k)\|^2_{\boldsymbol{\Lambda}}\} \tag{4}$$

subject to :

$$\boldsymbol{u}_{\min} \leq \boldsymbol{u}(k) \leq \boldsymbol{u}_{\max}$$
$$\Delta\boldsymbol{u}_{\min} \leq \Delta\boldsymbol{u}(k) \leq \Delta\boldsymbol{u}_{\max}$$
$$\boldsymbol{y}_{\min} \leq \hat{\boldsymbol{y}}(k) \leq \boldsymbol{y}_{\max}$$

where $\Delta\boldsymbol{u}_{\min}$, $\Delta\boldsymbol{u}_{\max}$, $\boldsymbol{u}_{\min}$, $\boldsymbol{u}_{\max}$, $\boldsymbol{y}_{\min}$, $\boldsymbol{y}_{\max}$ denote the constraints imposed on manipulated and controlled variables, respectively.

Generally, it is better to use the numerical algorithm, in which constraints are handled in a systematic and relatively easy way on the whole horizon. On the contrary, in the explicit implementation, the projection of control signals on the constraint set is applied to the already calculated control values. Therefore, the numerical algorithms can predict the need of constraint satisfaction in advance and control the process more gently. Unfortunately, the quadratic optimisation used in numerical implementation is quite time-consuming, as more complex operations have to be done (sequence of matrix operations in order to optimise the problem numerically). The C++ implementations use similar number of parameters, however the explicit implementation can be formulated as a simple sequence of additions and multiplications performed on floating point variables. Thus, the explicit algorithm can be applied to fast processes for which short sampling periods must be used.

3. Experimental setup

Software implemented fault injector (FITS) has been already used in various experiments, Gawkowski and Sosnowski (2003, 2005, 2007), Gawkowski et al. (2008), Sosnowski, Gawkowski and Lesiak (2004), Trawczyński, Sosnowski and Gawkowski (2008). It is based on standard Win32 Debugging API to control and disturb (by software emulation of a fault) the execution of the software application under tests. The fault injector FITS disturbs directly the tested application only within so-called *testing areas* (Gawkowski and Sosnowski, 2007), which limits the scope of disturbances only to the selected parts of the application. Here, the code of the controller and its data are contained in the dynamically loaded library (DLL on the x86 architecture under Win32 operating system) to achieve separation from the process model and the main loop of the application, simulating the whole control system. Moreover, the same binary code of the process simulator (compiled as a normal executable) is used during experiments with different implementations of the control algorithm. The basic components of the tested application, disturbed during the experiments (dashed box) as well as the process model (not disturbed during experiments) are shown in Fig. 2.

The tested application is also instrumented to send some measurement results (e.g. related to the values of internal variables, output signal deviations, and signalisation of failures detected by the controller itself) to the fault injector using user-defined messages (collected by FITS, see Gawkowski and Sosnowski, 2003, 2005, 2007; Sosnowski, Gawkowski and Lesiak, 2004; Trawczyński, Sosnowski and Gawkowski, 2008, and references therein).

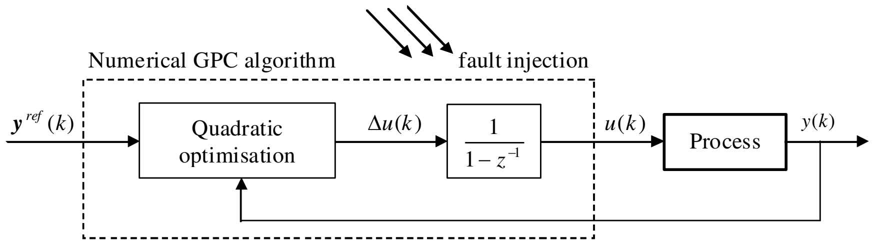

Figure 2. The structure of the control system with the numerical GPC algorithm

3.1. Process description

The process under consideration is a chemical reactor with two manipulated variables (u_1 and u_2, which correspond to the flow rate of the feed stream into the reactor and the flow rate of the cooling substance, respectively – these are outputs of the considered controller) and two controlled variables (y_1 – the concentration of the product, y_2 – the temperature in the reactor). It is described by the continuous-time transfer function model as in Camacho and Bordons (1999) (time constants in minutes):

$$\begin{bmatrix} Y_1(s) \\ Y_2(s) \end{bmatrix} = \begin{bmatrix} \dfrac{1}{1+0.7s} & \dfrac{5}{1+0.3s} \\ \dfrac{1}{1+0.5s} & \dfrac{2}{1+0.4s} \end{bmatrix} \begin{bmatrix} U_1(s) \\ U_2(s) \end{bmatrix}. \tag{5}$$

The explicit and numerical GPC algorithms have been designed. They use the discrete-time dynamic model obtained with the sampling period $T_p = 0.03$ minutes. In both cases the prediction horizon is $N = 3$, the control horizon $N_u = 2$, the weighting matrices are: $\boldsymbol{M} = \boldsymbol{I}$, $\boldsymbol{\Lambda} = \lambda \boldsymbol{I}$, where $\lambda = 0.075$. The magnitudes of manipulated variables are constrained: $u_1^{\min} = -2.5$, $u_1^{\max} = 2.5$, $u_2^{\min} = -0.6$, $u_2^{\max} = 0.6$, whereas the control increments and values of controlled variables are not constrained; 250 discrete time-steps of control system operation are simulated according to the following scenario:

- the process starts from a given set-point ($y_1 = 0$, $y_2 = 0$),
- at the sampling instant k=10, y_1^{ref} changes to 0.7 and y_2^{ref} to -0.5,

- at k=110, y_1^{ref} changes to -0.7,
- at k=160, y_2^{ref} changes to 0.5.

Moreover, unmeasured disturbances are added to both outputs of the process. It is assumed that the disturbances change in a sinusoidal way (inverted on the second output variable) with the amplitude equal to 0.2 and the period of 50 discrete time instants, i.e.

$$y_i(k) = (-1)^i 0.2 \sin(0.04\pi \cdot i)\,. \tag{6}$$

3.2. Fault injection policy

In this study the single bit-flip faults within CPU registers, target application data and machine instruction code (latched and non-latched, Gawkowski and Sosnowski, 2007; Trawczyński, Sosnowski and Gawkowski, 2008) are considered (Section 1). Faults are injected pseudorandomly in the execution time of the application under tests (AUT) and in space (bit position within disturbed resource, distribution over application's memory) to mimic Single Event Upset (SEU) effects, Anghel, Leveugle and Vanhauwaert (2005), Gawkowski and Sosnowski (2003), Sosnowski, Gawkowski and Lesiak (2004). At the fault triggering instant the AUT execution is suspended, the error is introduced within the AUT context (e.g. a bit-flip within AUT memory), and the AUT execution is resumed being monitored by the FITS (e.g. for any exceptions). As FITS uses Windows programming interface for debuggers, it can control the execution of the AUT with the resolution of particular execution instant of a single machine instruction, giving the full control over the time space of fault injections. It is also possible to correlate the fault injection instant and the target fault location with the source code of the AUT (basing on the mapping of machine instructions into the source code lines) and with the observed fault effects. One fault is injected per single run of simulation called a *test* later on (250 iterations of GPC as described in Section 3.1). A set of tests with the same fault location constitutes an *experiment*. In all experiments the average number of faults per fault location is 1000. At the end of each test run, the qualification of control performance is made. Additionally, the details on each test (injected fault details, logged data from the application under tests, etc.) are available. This provides very useful feedback information for the application developer to decrease fault sensitivity by software-based fault-hardening mechanisms, as it is illustrated in Section 4.

3.3. Qualification of experimental results

Control algorithms require complex analysis of the process behaviour, Gawkowski et al. (2008), Trawczyński, Sosnowski and Gawkowski (2008). The standard factor *SSE* (Sum of Squared Errors – calculated over y_1 and y_2 is used as a measure of correctness:

$$SSE = \sum_{k=1}^{n} \left((y_1^{ref}(k) - y_1(k))^2 + (y_2^{ref}(k) - y_2(k))^2 \right) \tag{7}$$

where n denotes the simulation horizon (n=250). Fault injection instants are limited to the first half of the simulation to allow any possible erroneous behaviour to be reflected in the SSE index. The reference SSE value for considered output trajectories is 9.4174 (obtained during non-faulty execution – the trajectories $y_1^{ref}(k)$, $y_2^{ref}(k)$ are described in Section 3.1). Because of the dynamic nature of the process, the SSE value is not equal to 0 as it takes some time to reach the desired reference output values. First experiments show that responses with SSE<15 can be qualified as correct ones. However, the threshold SSE value must be chosen arbitrarily by an expert.

The whole experiment is conducted by FITS automatically. At the end of the experiment (a set of simulation runs, each disturbed by a single fault in a selected fault location) summarized results are given. Five classes of test results are distinguished:

- C: correct behaviour (SSE<15),
- INC: incorrect (unacceptable) behaviour (SSE≥15),
- S: test terminated by the system due to an un-handled exception (related to detected errors, e.g. memory access violations, invalid opcodes, parity errors, Gawkowski and Sosnowski, 2001; MSDN Library, no date)
- T: timed-out test.
- U: the built-in error detection mechanism of the controller identified a critical situation and signalled inability of proper operation.

For brevity, they will be called: C, INC, S, T, and U-category *tests*.

The post-experimental analysis of fault effects requires detailed information on the faults injected and the application behaviour. Because FITS provides details about every test (simulated fault injection), the 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 also saves its outputs (here, simulation results, i.e. a set of process signals (u_1, u_2, y_1, y_2) in subsequent sampling instants) into separate files for each test (file names 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 (see Section 3.2). By changing the SSE threshold we can admit various levels of control quality. It is also possible to trace the whole control trajectory to check safety conditions, performance, etc.

4. Experimental results

Six different versions of the numerical GPC algorithm are analyzed in the following subsections – the basic one (version **A**) without any fault detection/tolerance mechanism and five increasingly advanced ones (fault-hardened versions **B**÷**F**). They implement some exception handling, assertions or state recovery techniques.

4.1. Non-hardened basic implementation

The basic implementation (referred to here as version **A**) takes 3942 assembly instructions (12 736 bytes of code). The analyzed scenario is realized by the execution of 2 360 990 assembly instructions (dynamic profile).

Fig. 3 presents the golden run trajectories for the fault-free execution. The given set-points are easily achieved and the influence of the sinusoidal disturbances (eq. (6) in Section 3.1) is compensated very efficiently.

The most fault sensitive resource of the controller is its code. In case of a latched fault, the probability of INC behavior was 7% and high SSE values were obtained. Fig. 4 presents the distribution of SSE values in subsequent ranges of SSE values within the INC category.

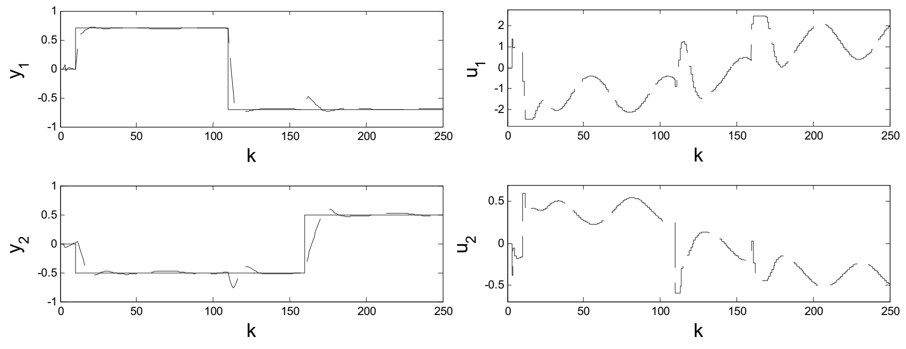

Figure 3. System responses for the fault-free golden run: set-point trajectory (*solid line*), incorrect behaviour (*dashed line*) SSE=9.4174

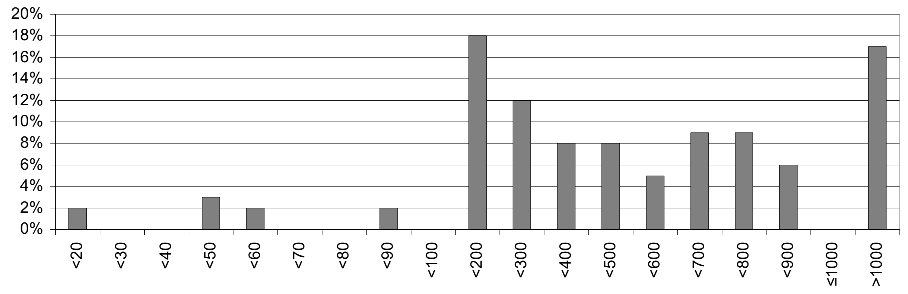

Figure 4. Distribution of SSE values (in ranges) within the INC-category (latched faults in code)

Even though the INC probability is quite low, the percentage of C-category tests was only 21.6%. Most of faults (also in case of other fault locations) resulted in controller termination by the operating system due to unhandled exceptions (S-category tests). So, after the visual analysis of INC-category test trajectories, several types of faulty process trajectories are identified. Fortunately, it is possible to detect such erroneous situations within the controller code. They are considered in the following subsections together with the step by step controller implementation improvement description. Subsequent dependability improvements are targeted at two aspects:

- integration with system error detection and handling mechanisms (here exceptions) to limit the percentage of S-category tests;
- inclusion of software based mechanisms to detect and further tolerate erroneous control.

Each of the mechanisms proposed is then also embedded in all subsequent versions. The summary of results for all considered versions is given at the end of this section (Fig. 8).

4.2. Fault tolerance through exception handling

As the basic version produces a lot of S-category test cases. It is clear that without proper exception handling any significant fault robustness will not be achieved. Moreover, the C++ statements *try/catch* (integrated with the native exception handling mechanism of the Win32 operating system, which is called Structured Exception Handling – SHE, Gawkowski et al., 2008; MSDN Library, no date) to handle exceptions can also be used as a framework for further error handling procedures. In version **B** of the analyzed application, the exception handling within the controller procedure resets the floating point unit as well as the parameters of the controller process model and applies to the process manipulated variables values computed in the previous iteration.

Experiments proved the efficiency of the proposed exception handling scheme in case of faults located within resources with states valid only during the single iteration of the control algorithm. The increase of C-category test percentage is significant: 67%, 30%, and 109% for faults located in CPU registers, data, and non-latched faults within instruction code, respectively. Unfortunately, the increase of INC-category is observed (see Fig. 8); in the worst case (non-latched faults in the code) it rises from 7% to 75.6%. The problem behind this is the lack of detection towards unrecoverable errors leading to repetitive exception handling execution in consecutive algorithm iterations.

4.3. Advanced exception handling

To overcome the drawback described above, in version **C** the exception handling procedure invocation counter is introduced. As the considered fault model assumes only single fault per application execution, only single exception handling attempt is allowed. The second exception (in further algorithm iterations) triggers a safety procedure, which signals the inability of the algorithm to continue the operation and it is followed by its self-termination (U-category tests – see Section 3.3). It is also possible to extend the proposed counter in real-life applications to be implemented e.g. as a multivalued health measure with a threshold defined and different incrementing/decrementing policy (e.g. resetting the counter after a few subsequent exception-free iterations), as described in Gawkowski and Sosnowski (2001) and the references therein.

Introducing the exception counter not only reduced significantly the probability of INC-category tests (more than ten times) for all fault locations compared to version **B** (see Fig. 8) but also in case of faults latched in the instruction code. In the case of non-latched faults in the code the INC cases are not observed at all. However, a typical unsafe scenario related to the violation of control value constraints is identified within the remaining INC cases, as shown in Fig. 5.

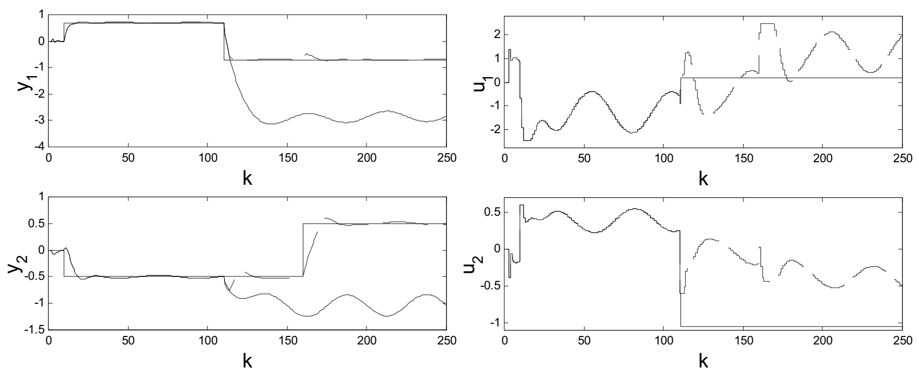

Figure 5. System responses to fault occurring at the 108th sampling instant at the algorithm data space; fault-free (*dashed line*), incorrect behaviour (*solid line*) *SSE*=852.8809

Responses shown in Fig. 5, obtained after the fault occurrence (solid lines) are clearly unsatisfactory. Both outputs are far from their set-point values. However, an early indication of bad operation of the controller is possible. The second control variable (u_2) violates its lower constraint since the 111th sampling instant as it stays on the value -1.0535, far beyond the constraint (-0.6), what results in a very high *SSE* value.

Another similar example of the violation of the bounds on control variables can be found in case of a fault located in the instruction code at the 104th sampling instant (instruction *xor esi, esi* changed to *xor esi, edi*). In this case, the value of u_2 in the 110th sampling instant is equal to -1.6507 (the constraint is violated by more than 1 – see Section 3.1) and in the next sampling instants it stays constantly at the value equal -0.6897 (also out of the range of constraints),

SSE=449.9906. It is interesting to note that similar results can be observed for different fault locations, here for data and instruction code. As the numerical algorithm should always generate the control variables within their limits, it is clear that the fault in the controller must have occurred.

4.4. Detection of control value violation

In version **D**, the suitable test against violations of control value constraints is introduced to the algorithm at the end of each iteration. Error detection triggers the exception occurrence that leads to the same handling procedure as in the previous version. This brought further improvement of the fault tolerance. Although the probability of INC cases is further reduced (see Figs. 8 and 9), unfortunately another dangerous behaviour can be observed (see Fig. 6).

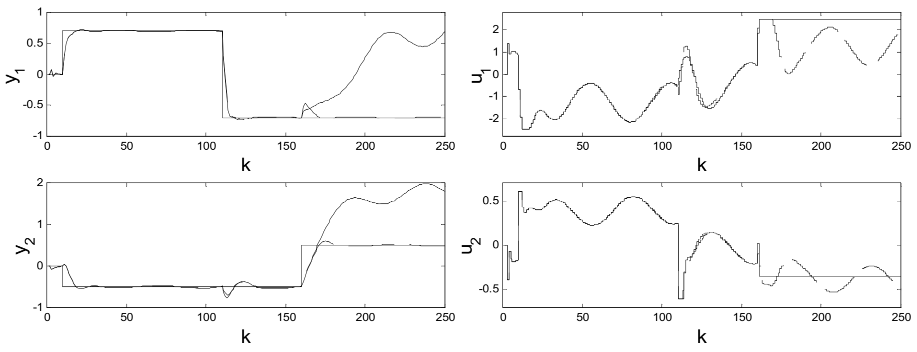

Figure 6. System responses to fault occurring at the 90th sampling instant within the instruction code; fault-free (*dashed line*), incorrect behaviour (*solid line*) SSE=199.9839

In the considered case both process outputs (y_1, y_2) are far from their set-point values like in examples discussed earlier. This is caused by the fact that both control variables are locked on constant values (sampling instances 157÷250). Contrary to the case of Fig. 5, though, this time both control variables are within their constraints (Section 3.1); in particular, the first control variable (u_1) is locked on the boundary. It should be stressed that in practice it is quite a common situation because usually, if the process is controlled efficiently, the values of control variables are equal to the values of constraints. Therefore, this situation not always indicates wrong behavior of the controller. On the other hand, in the studied case, the second control variable (u_2) is locked between its limits (not on the boundary). If both outputs are far from the set–point values, the controller should try to compensate them, hence, such a situation clearly indicates that something wrong has happened with the control algorithm. In the following part of this article, we refer to such a failure as a *locked controller*.

4.5. Locked controller detection

After introducing the suitable test for locked controller cases (detection triggers an exception) in version **E**, the detection rate of faulty situations improved even more with only less than 0.12% of dynamic overhead (compared to version D, see Fig. 9). Unfortunately, the remaining INC cases are hard to recognize with simple error detector procedures (see a sample case shown in Fig. 7).

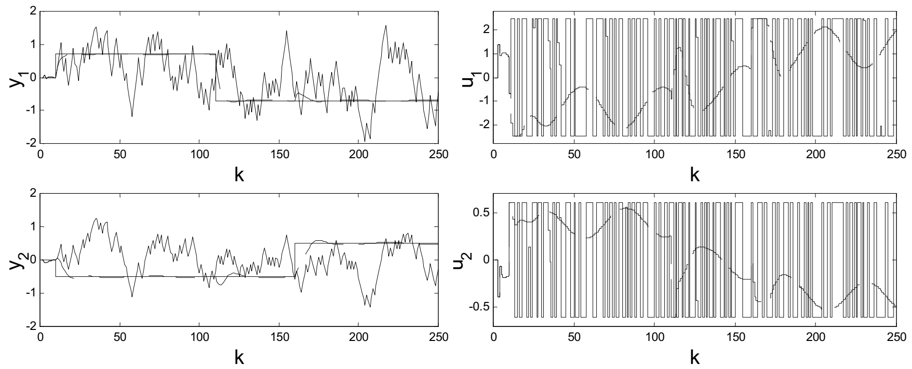

Figure 7. System responses to a fault that occurred at the 8th sampling instant; fault-free (*dashed line*), incorrect behaviour (*solid line*) SSE=312.3329

It is difficult to detect the faulty situation when both control variables are on limits, because this not always indicates improper behavior of the controller. Moreover, it should be emphasized that the shapes of output responses are not the result of the random disturbance, but of changes of control signal between minimal and maximal values, caused by the malfunction of the controller, corrupted by the injected fault.

4.6. Hybrid approach

It is worth noting that if both control values are within the constraint regions, the explicit GPC implementation has to produce the same control values as its numerical implementation. In practice however, deterioration of the obtained responses occurs frequently before control variables are locked or their values are on constraints. This can be observed, e.g., in Fig. 6, where differences between the faulty responses and the fault-free responses occur before the 100th sampling instant. Hence, error detection efficiency can be improved by the validation of the numerical GPC implementation with its explicit version (comparison of computed control variables values from both implementations – a hybrid approach – version **F**). Such an approach was found useful also in the case shown in Fig. 7, wherein the obtained responses are clearly unacceptable. Fortunately, in the 34th sampling instant, when control variables are between their limits, the hybrid approach detects the fault.

4.7. Summary of experiments

Fig. 8 summarizes experimental results for all considered versions (A÷F). For all examined fault locations the percentage of tests in each category is shown. As expected, the proposed hybrid solution (F) proved to be the most efficient one.

Fig. 9 depicts the relative efficiency of incorrect behaviour probability reduction compared to the basic version A (latched faults within the controller code). As the INC probability in version B rises dramatically (see Fig. 8 and Section 4.2), it is not considered here in order to keep the chart clear. Note that the very high degree of fault-tolerance is achieved at relatively low cost for all of fault-hardened versions (B÷F). Even if the static number of instructions in version F is by 26% higher than in version A, dynamically (related to the computation time) this overhead is only 18%, as the explicit GPC implementation is very compact compared to the numerical one (as mentioned in Section 2.2).

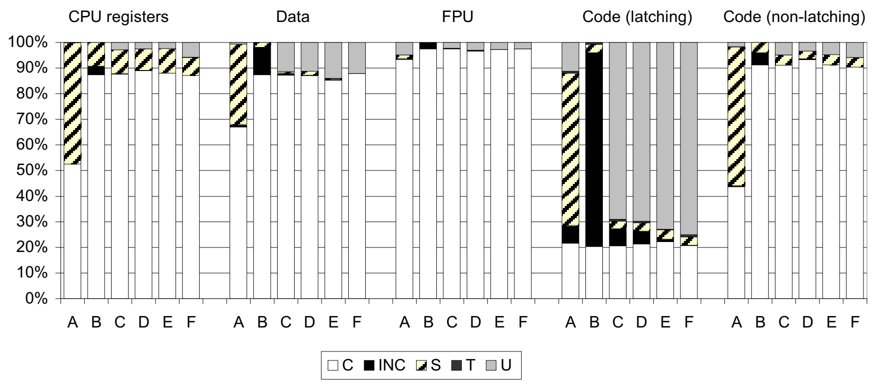

Figure 8. The summary of experimental results

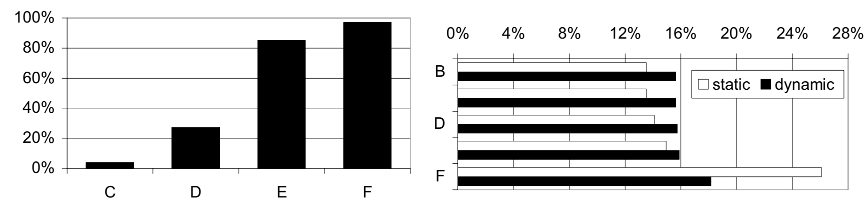

Figure 9. The relative INC-category probability reduction (*left*) and overhead costs compared to the number of instructions in version A (*right*)

5. Conclusions

The paper considers fault-hardening of software implementation of the numerical Generalized Predictive Control (GPC) algorithm. Simulation experiments have revealed that a large number of faults do not result in unacceptable control errors. A detailed analysis of the controller behaviour revealed various kinds of natural (intrinsic) redundancy. Some faults generate temporary (active for one or a few control iterations) deviations of control signals. Their effects are compensated in subsequent iterations, which may lead to some acceptable performance degradation of control. We have found that even in the case of sophisticated control algorithms involving quite long history of state variables, the natural fault tolerance capabilities are quite high, although they can be still increased significantly with simple software mechanisms. On the other hand, we have identified unstable behaviour of the system for some faults, which needs special treatment.

Six different versions of the algorithm are proposed and analysed (without and with some fault detection and fault tolerance mechanisms). First, the exception handling procedure is proposed and optimised. Then, several error detection mechanisms are added and integrated with the fault handling framework. The detection of the control value violations of the constraints in case of the numerical GPC is a natural choice. Next, the detection of the controller locking is introduced. Usually it is difficult to design a simple test detecting improper control system behavior during operation within prescribed boundaries, so, the hybrid verification extending the previous version is proposed. The additional verification of the calculated control values generated by the numerical GPC algorithm is performed using, as a reference, the explicit, compact GPC formulation. The latter concept is based on the fact that both algorithms should calculate identical control values if they lay within constraints boundaries.

The simulation experiments using our software implemented fault injector FITS clearly show that the fault-hardened implementation of the GPC algorithm offers significantly improved robustness with respect to faults. It is worth noting that the fail-bounded model provides better system productivity (higher probability of the operational mode) than the classical fail-silent model (here long non-operational mode in a safe state is admitted). On top of that, an excellent performance was obtained at a very low overhead (around 18%) of computation time.

Acknowledgment

This work was supported by the Polish national budget funds for science.

References

Anghel, L., Leveugle, R. and Vanhauwaert, P. (2005) Evaluation of SET and SEU effects at multiple abstraction levels. In: *Proc. 11th IEEE*

International On-Line Test Symposium. IEEE Press, 309–314.

Arlat, J. et al. (2003) Comparison of physical and software implemented fault injection techniques. *IEEE Transactions on Computers*, **52** (9), 1115–1133.

Baleani, M. et al. (2003) Fault-Tolerant Platforms for Automotive Safety-Critical Applications. In: *Proc. CASES 2003.* ACM, 170–177.

Camacho, E.F. and Bordons, C. (1999) *Model Predictive Control.* Springer.

Corno, F., Esposito, E., Reorda, M. and Tosato, S. (2004) Evaluating the effects of transient faults on vehicle dynamic performance in automotive systems. In: *Proc. ITC 2004.* IEEE Press, 1332–1339.

Cunha, J., Rela, M. and Silva, J. (2002) On the Use of Disaster Prediction for Failure Tolerance in Feedback Control Systems. In: *Proc. Dependable Systems and Networks 2002.* IEEE Press, 123–134.

Gaid, M. et al. (2006) Performance Evaluation of the Distributed Implementation of a Car Suspension System. In: *Proc. IFAC Workshop on Programmable Devices and Embedded Systems (PDeS 2006)*, IFAC.

Gawkowski, P., Ławryńczuk, M., Marusak, P., Sosnowski, J. and Tatjewski, P. (2008) Software Implementation of Explicit DMC Algorithm with Improved Dependability. In: T. Sobh, K. Elleithy, A. Mahmood, M.A. Karim, eds., *Novel Algorithms and Techniques in Telecommunications, Automation and Industrial Electronics.* Springer, London, 214–219.

Gawkowski, P. and Sosnowski, J. (2001) Experimental Evaluation of Fault Handling Mechanisms. In: U. Voges, ed., *SAFECOMP 2001.* **LNCS 2187**, Springer, 109–118.

Gawkowski, P. and Sosnowski, J. (2003) Dependability evaluation with fault injection experiments. *IEICE Transactions on Information & System*, **E86-D** (12), 2642–2649, December.

Gawkowski, P. and Sosnowski, J. (2005) Software implemented fault detection and fault tolerance mechanisms – part I: Concepts and algorithms. *Electronics and Telecommunications Quarterly*, **51** (2), 291–303.

Gawkowski, P. and Sosnowski, J. (2007) Experiences with software implemented fault injection. In: *Proc. International Conference on Architecture of Computing Systems.* VDE Verlag GmbH, 73–80.

Hecht, H. (1979) Fault-Tolerant Software. *IEEE Trans. on Reliability*, **R-28** (3), 227–232, August.

Isermann, R. (2006) *Fault-Diagnosis Systems: An Introduction from Fault Detection to Fault Tolerance.* Springer, Berlin-Heidelberg.

Korbicz, J., Kościelny, J.M., Kowalczuk, Z. and Cholewa, W. (2004) *Fault Diagnosis Models, Artificial Intelligence, Applications.* Springer, London.

Ławryńczuk, M., Marusak, M. and Tatjewski, P. (2008) Cooperation of model predictive control with steady-state economic optimisation. *Control and Cybernetics*, **37**, 133–158.

Maciejowski, J.M. (2002) *Predictive Control with Constraints.* Prentice Hall, Harlow.

Mariani, R., Fuhrmann, P. and Vittorelli, B. (2006) Fault Robust Microcontrollers for Automotive Applications. In: *Proc. 12th IEEE International On-line Test Symposium.* IEEE Press, 213–218.

Morari, M., Lee, J.H. (1999) Model predictive control: past, present and future. *Computers and Chemical Engineering,* **23** (4-5), 667–682.

MSDN Library (no date) *Structured Exception Handling.* http://msdn.microsoft.com/en-us/library/

Nouillant, F. et al. (2002) Cooperative Control for Car Suspension and Brake Systems. *J. of Auto. Tech.*, **4** (4), 147–155.

Qin, S.J., Badgwell, T.A. (2003) A survey of industrial model predictive control technology. *Control Engineering Practice*, **11** (7), 733–764.

Rossiter, J.A. (2003) *Model-Based Predictive Control.* CRC Press, Boca Raton.

Shooman, L. (2002) *Reliability of Computer Systems and Networks: Fault Tolerance, Analysis and Design.* John Wiley & Sons, New York.

Skarin, D., Karlsson, J. (2008) Software Implemented Detection and Recovery of Soft Errors in a Brake-by-Wire System. In: *Proc. 7th European Dependable Computing Conference*, 145–156.

Sosnowski, J., Gawkowski, P. and Lesiak, A. (2004) Fault injection stress strategies in dependability analysis. *Control and Cybernetics*, **33** (2), 679–699.

Tatjewski, P. (2007) *Advanced Control of Industrial Processes, Structures and Algorithms.* Springer, London.

Trawczynski, D., Sosnowski, J. and Gawkowski, P. (2008) Analyzing Fault Susceptibility of ABS Microcontroller. In: M.D. Harrison, M.A. Sujan, eds., *SAFECOMP2008.* **LNCS 5219**, Springer, 360-372.