On the Design of Robust Integrators for Fail-Bounded Control Systems

Jonny Vinter[1], Andreas Johansson[2], Peter Folkesson[1], Johan Karlsson[1]

[1]*Department of Computer Engineering*
Chalmers University of Technology
S-412 96 Göteborg, Sweden
+46 31 7721667, +46 31 7723663 fax
{vinter, peterf, johan}@ce.chalmers.se

[2]*Control Engineering Group*
Luleå University of Technology
S-971 87 Luleå, Sweden
+46 920 492334, +46 920 491558 fax
andreas.johansson@sm.luth.se

Abstract

This paper describes the design and evaluation of a robust integrator for software-implemented control systems. The integrator is constructed as a generic component in the Simulink design tool, and can thus be used for robust implementation of a wide range of control algorithms. The integrator is designed to support the fail-bounded failure model for transient bit-flips that may occur in the CPU, main memory and I/O circuits of a control system. In particular, it allows the control system to detect and recover from bit-flips that cause data errors. Robustness is achieved by sequentially executing duplicated integrator code on the same processor to support error detection, and through the use of a recovery buffer that allows a roll-back to the previous integrator state when an error is detected. The effectiveness of the robust integrator was evaluated through fault injection experiments with a PI controller, where single bit flips were injected inside the CPU of the control system. No violations of the fail-bounded model were observed in the experiments.

1. Introduction

One of the major challenges for designers of highly dependable control systems is to ensure that internal faults in the CPU, main memory, I/O interfaces and other parts of the controller do not cause the system to produce unacceptable or dangerous outputs. An increasingly important class of faults in computer hardware is radiation induced transient bit-flips, also known as soft errors. Particles such as heavy-ions, alpha particles and neutrons are known to cause soft errors in VLSI circuits [1]. So far, this has been a problem mainly for electronic equipment in airplanes, satellites and other aerospace applications. However, recent studies indicate that background radiation together with manufacturing residuals will cause significantly higher soft error rates in future VLSI devices also for ground-based systems [2, 3].

In this paper we present a software implementation of a generic integrator, which is robust with respect to soft errors. The integrator is constructed as a generic building block in the Simulink design tool, and can thus be used for the implementation of a wide range of dependable control algorithms. In particular, it allows the control system to detect and recover from bit-flips that cause data errors, which affect the state variables of the control algorithm [4]. The integrator is intended for systems that are fail-bounded [5, 6] for soft errors. This implies that the system should, in the presence of soft errors, produce either correct outputs, no outputs or incorrect outputs that have a minor or negligible impact on the controlled process.

The design requirements for a dependable control system typically include a failure model that the system must fulfill. One such model is the fail-silent model [7], which implies that the controller should produce either correct output or no output at all. Implementation of the fail-silence property is generally expensive as it requires perfect error detection coverage [8]. Many software-implemented techniques suitable for detecting data errors in fail-silent systems, for example code duplication [9, 10] or double execution [11] often features a code or time overhead of at least 100%.

Using the fail-bounded failure model is particularly suitable for control applications, and is less costly than using the fail-silent model. A fail-bounded control system is allowed to produce incorrect outputs provided the deviation in the output does not exceed a predefined bound. The bound is selected to ensure that the output errors have no significant impact on the controlled object.

The robust integrator detects data errors by a comparison of duplicated integrator state variables. The state variables are calculated in duplicated integrator code executed sequentially. When an error is detected, the integrator performs a best-effort recovery by loading a previous fault free state from a recovery buffer into the state variables. Thus, the design builds on the assumption that a roll-back by one sample interval is acceptable. The inputs to the integrator are protected by range checks based on the physical limitations of the controller inputs. The range checks ensure that the controller converges to a nominal behavior in a timely manner for errors caused by input disturbances and bit-flips in non-replicated input data. We have conducted a set of fault injection experiments in which we compare the effectiveness of the robust integrator with a conventional integrator in a PI control algorithm. Using scan-chain implemented fault injection, we injected single bit-flips inside the CPU that executed the control algorithms.

The remainder of the paper is organized as follows. Section 2 analyzes dependability weaknesses of controllers with respect to data errors, in view of a general controller structure. The design of the robust integrator, based on the analysis in Section 2, is presented in Section 3. The setup for the experimental evaluation of the integrator is described in Section 4, and the results of the evaluation are presented in Section 5. Finally, the conclusions are presented in Section 6.

2. A general controller

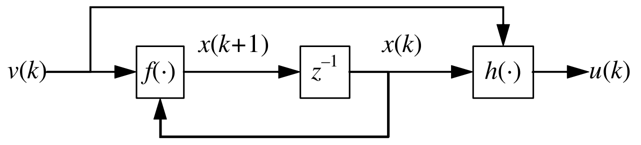

Figure 1. General controller structure.

Figure 1 is a graphical representation of a general controller $u(k) = h(x(k), v(k))$ and $x(k+1) = f(x(k), v(k))$. The input $v = [v_1\ v_2\ \dots\ v_p]^T$ is a vector that contains all the signals that are processed by the controller, i.e. both reference signals (set-points) and measurements (sensor values). The sensors measure the quantities that are subject to control. The output $u = [u_1\ u_2\ \dots\ u_q]^T$ of the controller consists of signals sent to the actuators of the controlled object, while the state $x = [x_1\ x_2\ \dots\ x_n]^T$ are the variables that need to be stored from one sampling instant to the next. The functions f and h may, in general, be nonlinear and z^{-1} represents the unit delay, i.e., delay from one sampling instant to the next. In this general structure, any controller can be represented, provided it has constant sampling rate and bounded dimension of the state vector. However, this representation does not specify e.g. execution order and is therefore not sufficient for uniquely representing the implementation of the controller.

All closed loop control systems have the ability to handle disturbances in the input signals, provided that the disturbances have a reasonable magnitude. Many bit-flips in the controller hardware that result in data errors have the same effect as pulse disturbances in the input signals. An obvious example of this is when a bit flip occurs in a variable that holds a sensor value. Assuming that the controller and the controlled process are linear, the effects of such disturbances converge to zero exponentially, since the closed loop system is exponentially stable. The assumption of linearity is valid if the magnitude of the disturbance is not too large. One way to ensure that the effects of disturbances converge to zero within an acceptable time is to introduce range checks on the inputs and outputs of the control system.

For most input signals v it is straightforward to associate an allowed range to each signal, i.e. $v_i(k) \in [\underline{V}_i, \overline{V}_i]$ for some constants $\underline{V}_i$ and $\overline{V}_i$. This range may, for a measurement input, be the range of the corresponding sensor while, for a reference signal, it is the set of allowed set points. In a similar manner, a range $[\underline{U}_i, \overline{U}_i]$ can often be associated to each output signal u_i, derived from the range of the corresponding actuator.

The impact of such range checks can be analyzed in the case of a linear controller and process using, e.g. the methods in [12] for state-space systems. The following analysis is for controllers in transfer function form. Assume a general 2-degrees-of-freedom controller $u(k) = F_r(z)r(k) - F_y(z)y(k)$ and a linear process $y(k) = G(z)u(k)$, where the vector y represents the measurement signals that are subject to control and the vector r contains the corresponding reference signals. When affected by disturbances, the produced output signal is (the arguments are excluded for readability), $u = F_r(r+\varphi_r) - F_y(y+\varphi_y) + \varphi_u$ where φ_r, φ_y, and φ_u are the disturbances that affect the set-point inputs, the measurement inputs, and the controller outputs, respectively. The closed loop transfer functions from the disturbance to the measurement can then be calculated as $y = G_c r + G_c\varphi_r - T\varphi_y + SG\varphi_u$ where $G_c = (I + GF_y)^{-1}GF_r$ is the servo system, $S = (I + GF_y)^{-1}$ is the sensitivity function and $T = (I + GF_y)^{-1}GF_y$ is the complementary sensitivity function.

In a properly designed control system, the above transfer functions have the following properties:

- T is the transfer function from measurement disturbance to measurement and is therefore designed to be small.
- G_c is the servo system and is generally ≈ 1 for low frequencies and ≈ 0 for high frequencies. In the common case of a 1-degree of freedom controller, $G_c = T$.

- *SG* is designed to be small due to the problem of control signal saturation.

In conclusion, a well-designed control system is inherently equipped to handle bit-flips that occur in the controller output and in the controller input provided that they are captured by a range check and thus bounded in magnitude.

Designing range checks for the state of the controller is difficult in the general case. Methods for observer-based fault detection could be utilized [13], but these tend to have a complexity that is comparable to, or even higher than, the complexity of the controller. Bit-flips in a state variable may result in an almost arbitrarily large erroneous value, compared to the correct state. Thus, it may take an unacceptably long time to converge to nominal behavior. Furthermore, nonlinear effects, e.g. actuator limits, may come into effect, preventing the system from converging at all. Consequently, we need to provide a mechanism, which ensures that the effect of errors in the state becomes bounded, and this mechanism must in the general case be more sophisticated than a range check.

In an implementation of a controller we distinguish between *local* and *global* variables and *constants*. The global variables store the state x, while the local variables and constants are required when calculating the functions f and h, or for holding the input and output signals during one sampling interval. Thus, local variables and constants are only used during one sampling interval, while the global variables store the state over time.

Below we give a few examples of errors that may lead to unbounded failures in the controller outputs:

- Errors in a local variable containing an element of the input v, which occur after the input range check has been executed.
- Errors that occur in a local variable or constant in the calculation of the function f.
- Errors that occur directly in a global variable holding an element of the state x.

Constants can be protected, for example by using checksums or by storing them in a protected memory. Since local variables in procedures or functions are re-created for each call, faulty values in those variables will only affect the controller output for one sampling instant and thus handled by the control system if captured by range checks. The same is true for global variables that are assigned new data each iteration of the control loop. The main problem is global variables that use their old (potentially faulty) values for calculating the new value, which may result in an erroneous state of the controller. Thus, the parts of a control algorithm most vulnerable to transient data errors, are the integrator state variables.

3. A generic robust integrator

In this section, we present the design of a robust integrator aimed at ensuring the fail-bounded property in the presence of bit-flips occurring in the controller hardware. The integrator handles bit-flips affecting the global state x. The bit-flips may affect the state either directly or indirectly through a local variable or constant in the function f. Note that bit-flips may permanently affect constants and that such faults are not handled by the robust integrator. However, as mentioned in Section 2, constants can be protected by checksums or by storing them in protected memory.

The generic structure enables the integrator to be used as a component in Simulink or any similar model-based design tool. The integrator is based on the Forward Euler method. We first discuss and compare the Forward Euler method with two other methods for implementing integrators, and then describe the implementation of the robust integrator.

3.1. Methods for implementing integrators

The Forward Euler method is also known as Forward Rectangular, or left-hand approximation. A realization of the Forward Euler method is shown as a block diagram in Figure 2. The integrator input is scaled by the sample interval T. The Unit Delay holds the state $x(k+1)$ from one sampling instant to another. In the implementation of the Forward Euler method it is important to consider the sequence of the operations represented in the block diagram. The sampling of the controller input is often controlled by a timer interrupt or an independent hardware unit to minimize the sampling jitter, and is thus executed independently of code that implements the control algorithm, including the integrator. The integrator code first sends the current state $x(k)$ to the integrator output so the algorithm can produce the controller output. Then the new state $x(k+1)$ is calculated by using the current state together with the integrator input calculated earlier in the control algorithm.

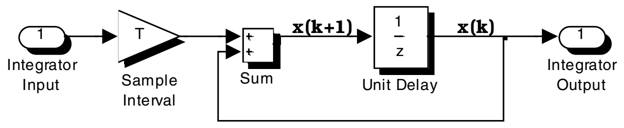

Figure 2. Realization of a Forward Euler integrator.

Other approaches for implementing integrators such as the Backward Euler and the Tustin (also known as Trapezoidal) methods uses a different execution order, as illustrated by the pseudo-code implementations for the simple controller below (see also Figures 1-2). Note that the integrator code is shown in bold.

Forward Euler

```
1. loop
2.    v(k) := read_controller_input
3.    u(k) := calculate_output(x(k), v(k))
4.    actuate_system(u(k))
5.    integrator_input(k) := calculate_ii(v(k))
6.    x(k + 1) := x(k) + integrator_input(k) * T
7.    k := k + 1
8.    wait_until_next_sample
9. end loop
```

Backward Euler

```
1. loop
2.    v(k) := read_controller_input
3.    integrator_input(k) := calculate_ii(v(k))
4.    x(k + 1) := x(k) + integrator_input(k) * T
5.    u(k) := calculate_output(x(k + 1), v(k))
6.    actuate_system(u(k))
7.    k := k + 1
8.    wait_until_next_sample
9. end loop
```

Tustin

```
1. loop
2.    v(k) := read_controller_input
3.    integrator_input(k) := calculate_ii(v(k))
4.    x(k + 1) := x(k) + integrator_input(k) * T/2
5.    u(k) := calculate_output(x(k) + x(k + 1), v(k))
6.    actuate_system(u(k))
7.    k := k + 1
8.    wait_until_next_sample
9. end loop
```

Which implementation method to choose for the integrator, depends on the application. However, a major advantage of using the Forward Euler method compared to the Backward Euler or the Tustin method is that the integrator output (i.e. the state $x(k)$) calculated in the previous sampling interval can be used immediately for calculating the controller output. The controller can therefore actuate the system before the state is updated (in pseudo-code line 4 instead of in line 6). The computational delay in the control algorithm is minimized and thereby also the overall possible *control delay*. A control delay is the time between related sampling and actuation actions.

Timing problems in real-time control systems [14] can arise from i) control delays (e.g. computational delays and network delays), ii) jitter (i.e. time-variations in actual start times of actions) and iii) transient errors. This paper addresses the task of handling transient data errors in control systems. We use software-implemented error detection and recovery in a robust integrator which introduces additional computational delay. By using the Forward Euler method for the robust integrator, we minimize the computational delay.

3.2. The robust integrator

The fault models assumed for the robust integrator are:

- Single or multiple transient bit-flips affecting not more than *one* of the two duplicated integrator states, concurrently.
- Transient erroneous controller inputs (e.g. sensor noise or a non-valid reference signal).

Figure 3 shows a Simulink block diagram of the robust integrator. It consists of one primary and one secondary integrator block as well as two *rc* blocks that implement the input range checks based on the physical limitations of the controller inputs, e.g. sensor ranges or allowed set-points. Errors detected by an *rc* block are handled by setting its output to zero and thereby disabling the integration for one sample. This operation acts as an impulse disturbance at the input, which the controller, according to the analysis in Section 2, is able to handle. The comparator block *c* detects errors in the state variables by comparing the states $x_1(k)$ and $x_2(k)$. If no error is detected, the state $x_1(k)$ is passed through the switches to the integrator output. Otherwise, the previous fault-free state $x_4(k)$ is fetched from a recovery buffer and used as output. Note from Figure 3 that in the fault-free case the state $x_3(k)$ has the same value as $x_1(k)$, and that $x_4(k)$ has the previous value of $x_1(k)$, namely $x_1(k–1)$. The primary and secondary block in Figure 3 has separate feed-back connections to prevent error propagation between the two blocks.

One incorrect output value with an arbitrary magnitude, lasting for one sample, is allowed for the integrator, usually limited at the output of the control algorithm or by physical limitations of the actuators. Thus, we do not use a range check on the integrator output. It is assumed that the sampling time is chosen short enough so that a saturated control signal, with a duration of one sample does not have serious consequences, e.g. brings the system to instability. Satisfying this condition is facilitated by the inherent low-pass filter characteristics of actuators, which filters out sudden changes in the control signal. The control signal delay, brought upon by a rollback recovery, can be regarded as a disturbance with a duration of one sample. This situation is therefore guaranteed to be acceptable if the above condition on the sample time is satisfied.

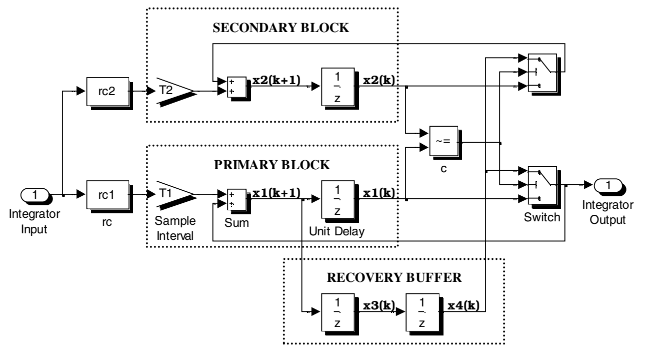

Figure 3. The robust integrator.

All components in Figure 3 are vectorized, meaning that the robust integrator can be used as a SISO (single input - single output) or MIMO (multiple input - multiple output) integrator.

By defining a comparison function

$$c(x,y,z) = \begin{cases} x, & x = y \\ z, & x \neq y \end{cases}$$

and a range check function, where L and U denotes the lower and upper range limit respectively

$$rc(i,L,U) = \begin{cases} i, & L \leq i \leq U \\ 0, & \text{otherwise} \end{cases}$$

the robust integrator can be expressed by the general controller structure in Figure 1 by defining the state vector as $x = [x_1 \;\; x_2 \;\; x_3 \;\; x_4]^T$ and the functions *f* and *h* as

$$f(x,i) = \begin{bmatrix} rc_1(i,L_1,U_1)T_1 + c(x_1,x_2,x_4) \\ rc_2(i,L_2,U_2)T_2 + c(x_2,x_1,x_4) \\ rc_1(i,L_1,U_1)T_1 + c(x_1,x_2,x_4) \\ x_3 \end{bmatrix}$$

$$h(x,i) = c(x_1,x_2,x_4)$$

Note that the sample interval $T_1 = T_2$, the lower range limit $L_1 = L_2$ and upper range limit $U_1 = U_2$, i.e., the constants *T*, *L* and *U* are duplicated.

Since the representation above does not specify the execution order for the robust integrator within a control algorithm, the corresponding pseudo-code for calculating the controller output and the new integrator states $x_1(k+1)$ and $x_2(k+1)$ is also given. The code that implements the robust integrator is given in bold.

```
-- perform a range check on the integrator input
function rc1(input, lower_limit, upper_limit)
    if (input >= lower_limit) and (input <= upper_limit)
        return (input) else return (0.0)

function rc2(input, lower_limit, upper_limit)
    if (input >= lower_limit) and (input <= upper_limit)
        return (input) else return (0.0)
```

```
-- compare states, actuate system and update states
loop
    v(k) := read_controller_input
    if x1(k) = x2(k) then
        u(k) := calculate_output(x1(k), v(k))
        actuate_system(u(k))
        x1(k+1) := x1(k) + rc1(integrator_input(k), L1, U1) * T1
        x2(k+1) := x2(k) + rc2(integrator_input(k), L2, U2) * T2
    else
        u(k) := calculate_output(x4(k), v(k))
        actuate_system(u(k))
        x1(k+1) := x4(k) + rc1(integrator_input(k), L1, U1) * T1
        x2(k+1) := x4(k) + rc2(integrator_input(k), L2, U2) * T2
    end if
    x3(k+1) := x1(k+1)
    x4(k+1) := x3(k)
    k := k + 1
    wait_until_next_sample
end loop
```

3.3. Recovery buffer

A Triple Modular Redundancy (TMR) system of integrators would theoretically have been able to detect data errors and make a true (exact) recovery. But if the state used for calculating the new states is set to a faulty value *after* the comparison, the fault will propagate into all three replicated states when they are being updated (via their closed loop connections, see for example Figure 2). The comparison in the following iteration of the control loop will not detect the fault since all three states will have the same faulty value.

Instead we use a duplex system with a recovery buffer which will not perform a true recovery but still is sufficient according to the fail-bounded model and have the following advantages compared to a TMR system:

- Less space and time overhead.
- Ability of recover from errors *after* the comparison is made by use of the recovery buffer.
- The recovery buffer is extendable to tolerate subsequent transient faults.

When an error is detected by the comparison function *c*, the recovery block is switched in. The recovery block uses a unit delay buffer of size n + 1, where n is the number of subsequent transient errors tolerated for the state $x_1(k+1)$. Therefore by using a buffer with size two, the output of the recovery block can not be updated erroneously if the assumption is that the same signal can not be corrupted in two subsequent samples. For example, to be able to tolerate a semi-permanent fault that directly affects the state $x_1(k+1)$ during two subsequent samples, we increase the buffer size to three, etc. The maximum allowed size of the buffer depends on the control application. After recovery, n fault-free samples are required to refresh the buffer. In our experiments and in the remainder of this paper, a buffer of size two is used.

Table 1 shows how transient data errors will be handled by the robust integrator for sample k. Observe that if one fault occurs in the state $x_1(k)$ *after* the comparison, the robust integrator will return a transient faulty state value that will be limited at the output of the controller or by actuator limits. Since the feed-back connections to the primary and secondary integrator blocks now will have different values, the state $x_1(k+1)$ will be updated with a faulty value but $x_2(k+1)$ will be updated correctly. This error is detected by the comparison function c in the next sample and a recovery is then performed by using the fault-free state $x_4(k)$.

Table 1. Internal error detection and recovery in the robust integrator.

	Leading to recovery in sample k	Leading to recovery in sample $k := k+1$	Integrator output in sample k	Integrator output in sample $k := k+1$
$x_1(k)$ is affected *before* comparison	✓		$x_4(k)$	$x_1(k)$
$x_1(k)$ is affected *after* comparison		✓	$x_1(k)$*	$x_4(k)$
$x_2(k)$ is affected *before* comparison	✓		$x_4(k)$	$x_1(k)$
$x_2(k)$ is affected *after* comparison		✓	$x_1(k)$	$x_4(k)$

* An undetected faulty value is given as output.

4. Experimental evaluation

4.1. Target application

Most controllers in the industry are based on PI or PID controllers. The derivative part of a PID controller, assuming an Backward Euler approximation, is calculated as $e(k) - e(k-1)$, where e denotes the control error. Therefore, the derivative part is less important to protect, since a bit-flip error that corrupts the control error will affect the controller output during at most two sample intervals and with opposite signs. However, sometimes the derivative part of the controller is filtered, in order to reduce the increased noise due to the differentiation. In this case, it may be desirable to protect the state variables of this filter in a manner similar to the protection of the integrator state.

We have used an engine control system based on a PI controller to investigate the effectiveness of the robust integrator. The engine control system was taken from an example library of Simulink and the PI controller was modified to use a constant sampling interval. Some continuous blocks were also converted into discrete blocks, facilitating the use of an Ada coder [15], generating the source code for the experimental evaluation. Figure 4 shows the block diagram of the engine control system highlighting the PI controller.

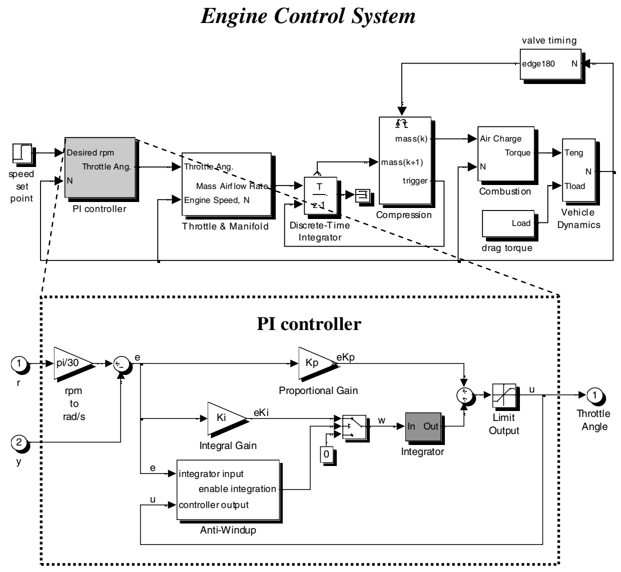

Figure 4. The PI controller.

The PI controller is equipped with an anti-windup function that is defined by

$$w(e,u) = \begin{cases} 0, & ((e \le 0) \wedge (u \le \underline{U})) \vee ((e \ge 0) \wedge (u \ge \overline{U})) \\ eK_i, & \text{otherwise} \end{cases}$$

where $\underline{U}$ and $\overline{U}$ are the lower and upper valid limit (stated in the saturation block *Limit Output* as well as in the *Anti-Windup* block) for the controller output u, and where e denotes the control error. For a limited output u, the anti-windup function will examine the sign of e to determine if the integration should be disabled or not to prevent integrator windup. This function also serves as an error detection and recovery mechanism for some data errors affecting the control algorithm. For example if the control error e is set to an erroneous value resulting in a limited output u (that is used as an argument for the anti-windup function) the anti-windup will be activated and the integrator input protected (set to zero). But if e is affected after the calculation of u, the anti-windup will not be activated and the integrator input will be set to eK_i, thus affected by the erroneous value of e. However, the robust integrator will detect and handle erroneous integrator inputs that are not captured by the anti-windup function.

4.2. Defining valid ranges

The input to the robust integrator is protected by the range check function *rc* in Figure 3. The lower and upper limit (L and U) are calculated based on the physical limitations of the sensor and reference values. For the engine control system these limits were calculated as follows. Assume that the engine control system is part of a car cruise control system and the engine speed sensor has a valid range between 0 and 10 000 rpm. By allowing the driver of the car to demand a momentary speed change of ±10 000 rpm (which are rather generous limits in this particular example application), the upper and lower limits of valid inputs to the integrator can be determined. The range is transformed to a range set in [rad/s] and scaled by the integral gain constant $K_i = 0.0723$ (see Figure 4) which results in an integrator input within the interval $\pm 10\,000[\text{rpm}] * \pi/30 * 0.0723 \approx \pm 76[\text{degrees/s}]$, i.e. $L = -76$ and $U = 76$. This range is used in our experiments for the two *rc* blocks shown in Figure 3. Observe also from Figure 3 that the integrator input is scaled by the sample interval $T = 0.012$[s]. If, for example, the fault-free integrator input equals the valid limit of 76[degrees/s] and the sign bit is affected by a bit-flip, this will result in a change to -76.0[degrees/s]. The maximal contribution to the integration is therefore: $2 * \pm 76[\text{degrees/s}] * 0.012[\text{s}] = \pm 1.82[\text{degrees}]$.

The range used for the Limit Output block (and the Anti-Windup block) is $\underline{U} = 0$, and $\overline{U} = 70$, due to physical limitations of the engine actuator.

4.3. Experimental setup

For the experimental evaluation, scan-chain implemented fault injection with the GOOFI tool [16] was used. Two versions of the PI controller, one with the nominal integrator shown in Figure 2 and the other with the robust integrator shown in Figure 3, executed on an evaluation board featuring the Thor microprocessor [17]. The evaluation board was connected to the expansion port of a Sun workstation which executed a simulation of the controlled engine (see Figure 4). For each experiment, 125 control loop iterations (corresponding to 1.5 s) were simulated, and one single bit-flip fault was injected randomly into the data cache or one of the internal- or programmer visible CPU registers of Thor, uniformly distributed in space and time. 5063 experiments were conducted with the PI controller using the nominal integrator and 4183 experiments with the PI controller using the robust integrator.

5. Results

5.1. Classification of the results

One way to quantify the results of fault injection experiments is to utilize the ℓ_1-norm and ℓ_∞-norm [12], defined by:

$$\|h\|_1 = \sum_k |h(k)| \quad \text{and} \quad \|h\|_\infty = \sup_k |h(k)|$$

respectively.

The norm $\|h\|_1$ is the sum of the absolute values over time for the signal h. The norm $\|h\|_\infty$ is the maximum absolute value of the signal h. Let u^0 and y^0 denote the control signals and the measurement signals in the nominal, fault-free case and let u^f and y^f be the same signals for the faulty case. The control errors due to the fault are then defined as $e_y = y^0 - y^f$ and $e_u = u^0 - u^f$. Furthermore, let $\mathbf{K} = \{k_{\text{first}}, k_{\text{first}} + 1, \ldots, k_{\text{last}}\}$ be the time interval of the simulation, where k_{first} and k_{last} are the first and the last observed sample number respectively. An incorrect output is said to have occurred if $e_u(k) \neq 0$ for some $k \in \mathbf{K}$.

The severity of the consequences of a fault depends a great deal on the process subject to control. However, one consequence that is always severe is instability. As mentioned in Section 2, this may happen if the actuator limits come into effect. In this case, the ℓ_1-norm of the error signals e_u and e_y are unbounded while the ℓ_∞-norm may be bounded, but only due to actuator limits. If the actuator stays unsaturated, and the control error e_u converges to zero in a controlled manner, the consequences may still be severe, depending on the magnitude of e_u and the time scale of its convergence. If the control error converges exponentially to zero, both the ℓ_1-norm and the ℓ_∞-norm of the control error are defined and provide valuable information on the severity of the fault. Classifying the effects of an incorrect output by whether or not the control error converges is, however, not feasible from a practical point of view, since this is, in general, impossible to determine from a simulation over finite time. Therefore, the following practically useful definitions are made.

- **Saturated incorrect output –** An incorrect controller output u^f was produced during the simulation period and is saturated for at least two samples. We consider this a violation of the fail-bounded property.

- **Non-saturated incorrect output –** An incorrect controller output u^f was produced but is not saturated more than one sample, which we consider as fail-bounded.

The severity of the incorrect outputs in the *Non-saturated incorrect output* class is ranked by the ℓ_1-norm and ℓ_∞-norm of the control error. Since an experiment can only be performed for a limited period of time, norms can not be determined from the experimental results and therefore the following approximations are used instead

$$\|h\|_1 \approx \sum_{k\in\mathbf{K}} |h(k)| \text{ and } \|h\|_\infty \approx \sup_{k\in\mathbf{K}} |h(k)|$$

The mean value of the above norm approximations over all experiments in the *non-saturated incorrect output* class, together with the percentage of *saturated incorrect outputs* are used to compare the PI controller with the nominal integrator, and the PI controller with the robust integrator in Section 5.4.

5.2. PI controller with the nominal integrator

Figure 5 shows how the engine control example presented in section 4.1 responds to a saturated incorrect controller output. Note that the engine would be saturated even if the incorrect controller output had not been saturated by the Limit Output block in Figure 4, since the engine actuator is also limited. However, the system is now non-linear because of saturation and will not converge exponentially as would have been the case for the linear system. Saturated incorrect outputs were observed for more than 8% of the experiments leading to erroneous controller outputs with the PI controller using the nominal integrator (corresponding to 0.3% of all the experiments).

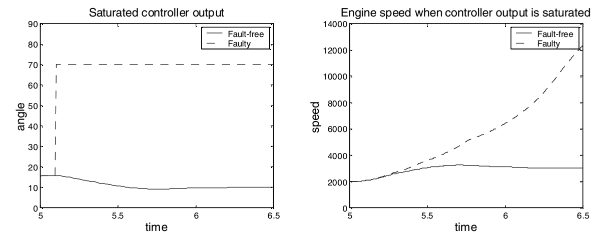

Figure 5. Fault-free vs. faulty controller output and the resulting engine speed when the controller output is saturated.

5.3. PI controller with the robust integrator

The robust integrator is designed to handle internal data errors affecting the state of the integrator as well as non-valid inputs to the integrator. Saturated incorrect outputs, such as the one shown in Figure 5, were not observed in any of the experiments with the robust integrator. However, transient incorrect outputs with arbitrary magnitudes, lasting for one sample are allowed for the integrator. If the magnitude of this output exceeds the actuator limits, the output is saturated by the actuator. Figure 6 shows how such a transient incorrect output will affect the engine.

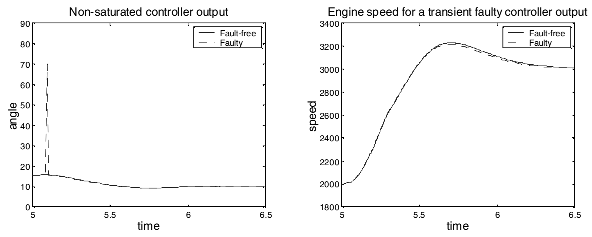

Figure 6. Fault-free vs. faulty controller output and the resulting engine speed for a transient faulty controller output.

In Section 4.2, the valid limits for the integrator input were calculated to ± 76[degrees/s]. Figure 7 and Figure 8 show the fault-free and faulty output of the engine controller together with the engine speed for an incorrect input equal to the valid limits of ± 76[degrees/s]. This is the worst observed behavior of the PI controller with the robust integrator.

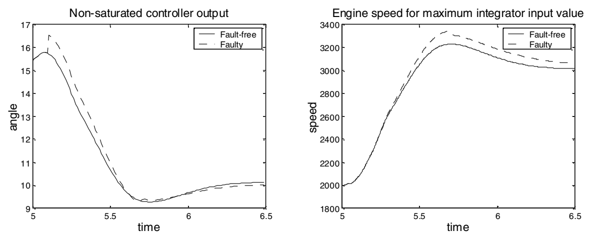

Figure 7. Fault-free vs. faulty controller output and engine speed when the integrator input assumes a transient *maximum* valid value = 76.

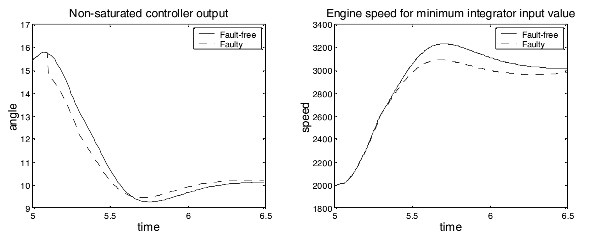

Figure 8. Fault-free vs. faulty controller output and engine speed when the integrator input assumes a transient *minimum* valid value = - 76.

5.4. Comparison

Results from the experiments with the nominal and the robust integrator are presented and compared in Table 2. 73.7% and 72.7% of all injected faults resulted in correct outputs, i.e. latent or overwritten errors, for the nominal and robust integrator respectively. The hardware EDMs of the Thor CPU were triggered by 22.6% of the injected faults for the nominal integrator vs. 23.0% for the robust integrator. When an error was detected by the CPU, an exception was triggered causing a reset of the computer leading to the start of a new experiment. As mentioned in Section 5.2, 0.3% of the experiments resulted in saturated incorrect outputs for the nominal integrator corresponding to more than 8% of the experiments leading to erroneous controller outputs. Most of these faults locked the engine throttle at 70.0 degrees during the observed interval (1.5 s) as shown in Figure 5. No saturated incorrect outputs were observed for the robust integrator. The total percentage of non-saturated and saturated incorrect outputs observed for the nominal integrator is 3.7% vs. 4.3% for the robust integrator.

The use of the recovery buffer and the range checks in the robust integrator do not result in an exact recovery (see Section 3), resulting in a small deviation from the correct output. Thus, all data errors handled by the error detection and recovery mechanisms in the robust integrator resulted in non-saturated incorrect outputs.

If the PI controller was not equipped with the anti-windup function, more data errors would have resulted in saturated incorrect outputs for the nominal integrator but they would have been detected and handled by the robust integrator (see Section 4.1).

Table 2. Experimental results.

	PI controller with nominal integrator	PI controller with robust integrator
Correct output	73.7% (#3731)	72.7% (#3039)
Detected by hardware EDMs	22.6% (#1144)	23.0% (#964)
Saturated incorrect output	0.3% (#15)	0% (#0)
Non-saturated incorrect output	3.4% (#173)	4.3% (#180)
TOTAL	100% (#5063)	100% (#4183)

Table 3 shows the mean value of the ℓ_1-norm and the maximum observed ℓ_∞-norm of the errors, for the faults resulting in non-saturated incorrect outputs according to the approximations made in Section 5. These results show more than a twofold improvement using the robust integrator compared to the nominal integrator.

Table 3. Norms for the non-saturated class.

Norm	PI controller with nominal integrator	PI controller with robust integrator	Improvement ratio
$\|e_u\|_1$	0.19	0.08	2.38
$\|e_y\|_1$	23.10	4.80	4.81
$\|e_y\|_\infty$	1151.65 rpm	368.98 rpm	3.12

The overall characteristics of the two versions of the PI controller are compared in Table 4. The code and time overhead are presented normalized to the nominal version of the integrator. The code for the Anti-Windup function in the PI controller (see Figure 4) is quite large compared to the entire algorithm, which results in a code overhead when using the robust integrator of only 20%. The time overhead is also 20% for the PI controller with the robust integrator.

The worst observed behavior for the PI controller with the nominal integrator was saturated incorrect outputs as shown in Figure Figure 5, and with the robust integrator the behavior shown in Figures 6-8.

Table 4. Overall characteristics.

	PI controller with nominal integrator	PI controller with robust integrator
Space overhead	1	1.2
Time overhead	1	1.2
Worst observed behavior	Saturated incorrect output (e.g. locking the engine throttle) as shown in Figure 5	Non-saturated incorrect output as shown in Figure 6-8

6. Conclusions and future work

Starting from a general structure of a control algorithm, we have highlighted the weaknesses of control systems regarding sensitivity to internal computer errors. A solution is suggested and evaluated for a PI controller and can be generalized to more advanced controller structures. We have designed a robust integrator that protects its state from transient faults that may lead to instability of the closed loop system. The effectiveness of the robust integrator was evaluated through fault injection experiments with a PI controller, where single bit-flips were injected into the CPU of the control system. No violations of the fail-bounded model, defined as incorrect controller outputs saturated for more than one sample, were observed for more than 4000 experiments with the robust integrator. When using a nominal unprotected integrator, the observed percentage of violations was more than 8% of the experiments leading to erroneous controller outputs.

Our software-implemented approach is more cost-effective in terms of code and time overhead compared to many other techniques since only the state of the control algorithm is protected, instead of the entire algorithm. The generic design of the robust integrator facilitates its use in a model-based design tool for implementation of dependable control algorithms.

Our future work will focus on investigating software-implemented fault handling for multiple-input and multiple-output control algorithms, such as jet-engine controllers. We will design and experimentally validate enhancements of the robust integrator for use in distributed jet-engine controllers.

Acknowledgements

This work was financed by NFFP (Swedish National Flight Research Program). We want to specially thank Torbjörn Norlander and Olof Hannius at Volvo Aero Corporation, and Torbjörn Hult and Stefan Asserhäll at Saab Ericsson Space AB, for valuable suggestions and technical assistance.

References

[1] G.C. Messenger, "Collection of Charge on Junction Nodes from Ion Tracks". *IEEE Transactions on Nuclear Science*, 1982. ns-29(6): pp. 2024-2031.

[2] C. Constantinescu. "Impact of Deep Submicron Technology on Dependability of VLSI Circuits". in *Proceedings International Conference on Dependable Systems and Networks*. 2002. Los Alamitos, CA, USA: IEEE Comput. Soc.

[3] P. Shivakumar, M. Kistler, S.W. Keckler, D. Burger, and L. Alvisi. "Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic". in *Proceedings International Conference on Dependable Systems and Networks*. 2002. Los Alamitos, CA, USA: IEEE Comput. Soc.

[4] J. Vinter, J. Aidemark, P. Folkesson, and J. Karlsson. "Reducing Critical Failures for Control Algorithms Using Executable Assertions and Best Effort Recovery". in *Proceedings International Conference on Dependable Systems and Networks. 1-4 July 2001*. 2001. Göteborg, Sweden: Soc Los Alamitos CA USA.

[5] J.G. Silva, P. Prata, M. Rela, and H. Madeira, "Practical Issues in the Use of ABFT and a New Failure Model". *Proceedings of 28th International Symposium on Fault Tolerant Computing*, 1998: pp. 26-35.

[6] J.C. Cunha, R. Maia, M.Z. Rela, and J.G. Silva. "A Study of Failure Models in Feedback Control Systems". in *Proceedings International Conference on Dependable Systems and Networks. 1 4 July 2001*. 2001. Goteborg, Sweden: Soc Los Alamitos CA USA.

[7] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, "Distributed Fault-Tolerant Real-Time Systems: The Mars Approach". *IEEE Micro*, 1989. 9(1): pp. 25-40.

[8] T.F. Arnold, "The Concept of Coverage and Its Effect on the Reliability Model of a Repairable System". *Digest of Papers from the 1972 International Symposium on Fault-Tolerant Computing*, 1972: pp. 200-204.

[9] A. Benso, S. Chiusano, P. Prinetto, and L. Tagliaferri. "A C/C++ Source-to-Source Compiler for Dependable Applications". in *Proceedings of International Conference on Dependable Systems and Networks (includes FTCS 30 30th Annual International Symposium on Fault Tolerant Computing and DCCA 8). 25 28 June 2000*. 2000. New York, NY, USA: Soc Los Alamitos CA USA.

[10] M. Rebaudengo, M. Sonza Reorda, M. Torchiano, and M. Violante. "Soft-Error Detection through Software Fault-Tolerance Techniques". in *1999 Proceedings IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems. 1 3 Nov. 1999*. 1999. Albuquerque, NM, USA: Soc Los Alamitos CA USA.

[11] B.W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*. Addison-Wesley Series in Electrical and Computer Engineering,. 1989, Reading: Addison-Wesley. xviii, 584 s.

[12] Ö. Askerdal, M. Gäfvert, M. Hiller, and N. Suri. "A Control Theory Approach for Analyzing the Effects of Data Errors in Safety-Critical Control Systems". in *Proceedings Pacific Rim International Symposium on Dependable Computing. 16-18 December 2002*. 2002. Tsukuba Science City, Japan.

[13] P.M. Frank and X. Ding, "Survey of Robust Residual Generation and Evaluation Methods in Observer-Based Fault Detection Systems". *Journal of Process Control*, 1997. 7(No. 6): pp. 403-424.

[14] B. Wittenmark, J. Nilsson, and M. Törngren. "Timing Problems in Real-Time Control Systems". in *Proceedings of 1995 American Control Conference - ACC'95*. 1995. Evanston, IL, USA: American Autom Control Council.

[15] http://www.mathworks.com.

[16] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. "GOOFI: Generic Object-Oriented Fault Injection Tool". in *Proceedings International Conference on Dependable Systems and Networks. 1 4 July 2001*. 2001. Goteborg, Sweden: Soc Los Alamitos CA USA.

[17] Saab Ericsson Space AB. "Microprocessor Thor, Product Information". 1993.