Error Recovery Using Forced Validity Assisted by Executable Assertions for Error Detection: An Experimental Evaluation*

Martin Hiller
Department of Computer Engineering
Chalmers University of Technology
SE-412 96, Göteborg, SWEDEN
hiller@ce.chalmers.se

Abstract

This paper proposes and evaluates error detection and recovery mechanisms suitable for embedded systems. The purpose of these mechanisms is to provide detection of and recovery from data errors in internal variables. A classification scheme for variables enables us to construct a parameterised detection and recovery mechanism for each variable. Error detection is handled by executable assertions and recovery is attempted by forcing an erroneous variable into the valid domain of that variable. The effect on failure probability from some of the proposed mechanisms is evaluated in an error injection experiment targeting an embedded system. Errors were injected into the original system without mechanisms as well as into an instrumented system. Results show that the failure probability decreased most for errors in monitored variables and very little for errors in variables that were not directly monitored. For errors in the stack and CPU registers, no significant change was observed.

1. Introduction

In the last decade or two, computers have found their way into systems aimed mainly at the consumer market. For instance, modern cars have a large number of embedded control systems handling many of the electrical functions in the car. More and more safety-related functions are controlled by software, thereby making software reliability crucial in these systems. In addition, the high-volume production series of consumer systems demands low production cost as well as low maintenance cost. This, together with the requirements on high reliability, motivates the search for inexpensive, yet effective, techniques to improve the ability of software to cope with faults and errors.

A common way of coping with faults and errors using software is to deploy multiple, diverse versions of the software. These versions may be organised in a variety of structures such as, for example, N-version programming [1] or recovery blocks [2]. However, systems using such structures will, in most cases, be high-cost systems since multiple, functionally equivalent versions must be developed. Also, more powerful hardware is often needed, increasing the production cost. Therefore, such structures are most commonly found in systems that can carry a high cost level – *e.g.* the systems controlling aircraft, spacecraft or nuclear power plants. Also, Randell and Xu [3] state that "the overall success of recovery block schemes rests to a great extent on the effectiveness of the error detection mechanism used – especially on the acceptance tests". This makes the search for inexpensive error detection techniques valid also for structures like recovery blocks.

Error detection may be provided using on-line tests of internal data in the form of executable assertions [4]. Executable assertions test the validity of the value of a variable using predefined rules and can be used both during software development to aid developers in finding faults in the system [5] and when the system is operational as part of fault-tolerance mechanisms [6]. The main drawback of executable assertions is that they are highly application specific, meaning that in order to construct effective assertions, developers must have extensive knowledge of the target system. Studies have shown that the ability to develop effective assertions is highly individual among software developers [7]. By making the development of executable assertions a part of the normal system design process rather than a task that is performed when the system enters a test phase, or even worse, after the system has been made operational, may decrease the effect of differences between individuals.

* This research was supported by Volvo and by NUTEK under contract 1P21-97-4745.

Rabéjac presented a development method for executable assertions [6]. Unfortunately, no in-depth description of this method was provided. Stroph and Clarke presented dynamic acceptance tests [8], which are executable assertions with dynamic constraints. However, their proposed scheme applies only to linear, causal, time-invariant systems. Many systems, though, may not be time-invariant, and therefore require other measures.

This paper proposes a method for systematically classifying the internal data signals in an embedded control system. A specific set of validity constraints is assigned to each signal class in this classification. These constraints are used to detect errors and to recover the signal from these errors. The proposed recovery mechanism is called *forced validity*, as it maintains the validity of the internal data by forcing erroneous signals into their respective valid domain.

In order to evaluate the detection and recovery capability of the proposed mechanisms, a case study on an embedded control system used for arresting aircraft was carried out. The case study aimed at assessing the impact of the proposed mechanisms on failure behaviour induced by internal data errors. Although both hardware faults and software faults may induce such errors, the case study concentrates on errors induced by hardware faults.

Section 2 contains a description of the proposed classification scheme and the executable assertions and forced validity recovery mechanisms. Section 3 describes the case study. The results of the experiments are shown in section 4. Section 5 consists of a discussion of the obtained results and section 6 summarises the study and proposes future work.

2. Error detection and recovery

Error detection in the form of executable assertions can potentially detect any error in internal data caused by either software faults or hardware faults [7]. When input data arrive at a functional block (*e.g.* a function or procedure), they are subjected to executable assertions determining whether they are acceptable. Output data from calculations may also be tested to see if the results seem acceptable. Should an error be detected, measures can be taken to recover from the error, and the signal can be returned to a valid state.

2.1. Signal classification

Constructing executable assertions and recovery mechanisms is very application specific. A rigorous way of classifying the data that are to be tested will help to determine the valid domain for the signals. This paper proposes a classification scheme to systemise the construction of executable assertions and recovery mechanisms (shown in Figure 1).

The two main categories in the classification scheme are continuous and discrete signals. These categories have subcategories that further classify the signal. Details are found below.

All signals in an embedded control system may be classified according to the proposed scheme. For every signal class we can set up a specific set of constraints, such as boundary values and rate limitations, which are then used in the executable assertions and recovery mechanisms. In order to enable a signal to have different behaviours during different modes of operation in the system, a signal may have one set of constraints for each such mode. Which constraints are to be used is defined by the current mode of the signal.

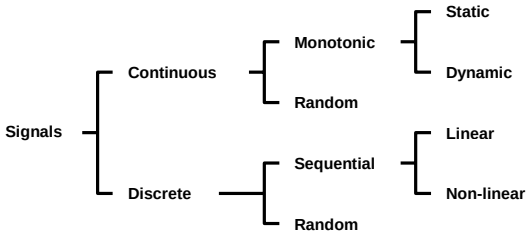

Figure 1. Signal classification scheme.

Error detection is performed as a test of the constraints. A violation of a constraint is interpreted as the detection of an error. Error recovery is achieved by forcing validity onto the erroneous signal. That is, the erroneous signal is set to a valid "best-effort" value based on the constraints of that particular signal and the previous (valid) value. The recovery mechanisms are separate from the classification scheme, meaning that other recovery mechanisms may be devised using the same classification.

Continuous signals. The continuous signals are often used to model signals in the environment that are of continuous nature. Such signals are typically representations of physical signals such as temperatures, pressures or velocities.

The continuous signals can be divided into *monotonic* and *random* continuous signals. Monotonic signals must either increase or decrease their value monotonically and cannot, for example, increase between the first and the second test and then decrease between the second and the third test. However, they may be allowed to remain unchanged between tests. The monotonic signals can have either a *static* rate or a *dynamic* rate. A signal with static rate must either increase or decrease its value with a given constant rate. A signal with dynamic rate, however, can change at any rate that is within the specified range. The random continuous signals may decrease or increase (or

remain unchanged) between tests (that is, they may randomly increase or decrease between tests).

Also, a signal may be allowed to wrap around, *i.e.* when it has reached its maximum or minimum value, it may continue "on the other side". This is visualised in Figure 2, which shows examples of the three types of continuous signals.

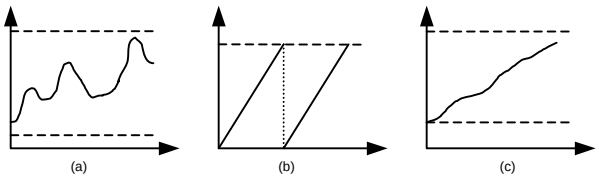

Figure 2. Continuous signals: (a) random, (b) static monotonic (with wrap-around), (c) dynamic monotonic

For the proposed error detection and recovery mechanisms (see section 2.2), we assign to each continuous signal a set P_{cont} containing seven different parameters: $s_{\max}$ (maximum value), $s_{\min}$ (minimum value), $r_{\min,incr}$ (minimum increase rate), $r_{\max,incr}$ (maximum increase rate), $r_{\min,decr}$ (minimum decrease rate), $r_{\max,decr}$ (maximum decrease rate), and w (wrap-around allowed/not allowed). Each of these signal classes imposes certain constraints on the parameters, as shown in Table 1.

Signal class	Parameters
All	$s_{\max} > s_{\min}$, w = allowed/not allowed
Static monotonic	($r_{\max,incr} = r_{\min,incr} = 0$, $r_{\max,decr} = r_{\min,decr} > 0$) or ($r_{\max,decr} = r_{\min,decr} = 0$, $r_{\max,incr} = r_{\min,incr} > 0$)
Dynamic monotonic	($r_{\max,incr} = r_{\min,incr} = 0$, $r_{\max,decr} > r_{\min,decr} \geq 0$) or ($r_{\max,decr} = r_{\min,decr} = 0$, $r_{\max,incr} > r_{\min,incr} \geq 0$)
Random	$r_{\max,incr} \geq r_{\min,incr} \geq 0$, $r_{\max,decr} \geq r_{\min,decr} \geq 0$

Table 1. Parameters for continuous signal classes.

For statically increasing monotonic signals the change rate limits for decrease are set to zero (*i.e.* $r_{\min,decr} = r_{\max,decr} = 0$) and the change rate limits for increase are set to the same value (*i.e.* $r_{\min,incr} = r_{\max,incr} > 0$). For a statically decreasing signal, instead the increase rate limits are set to zero and the increase rates are both set to the same value. For random continuous signals we have different values for the change rate limits (*i.e.* $r_{\min,incr} \neq r_{\max,incr}$ and/or $r_{\min,decr} \neq r_{\max,decr}$). These parameters are static, but dynamic constraints as in [9] may also be considered.

Discrete signals. Discrete signals are allowed to take on a set of discrete values. They often contain information on the settings on an operator panel or the operation mode of the system. Actually, all signals containing some kind of state information internal or external to the system may be classified as discrete signals. For instance, execution sequences that must be followed in a certain order, or state machines with a number of states and a number of transitions between the states, may be modelled as discrete signals. The discrete signals are divided into sequential and random signals.

A sequential signal has constraints on how it may change its value from any given other value, *i.e.* the order of change is restricted. They are divided into *linear* and *non-linear* signals. Linear signals must traverse their valid domain in a fixed predefined order, one value after another. For instance, the execution sequence mentioned above could be modelled as a linear signal. Non-linear signals traverse their valid domain in predefined ways. The random signals are allowed to make any transition from one value to another within the valid domain of the signal.

For the proposed error detection and recovery mechanisms (see section 2.2) we assign to each signal a set P_{disc} containing these parameters: D (the set of valid values), $T(d)$ (the set of valid transitions from element d in D; there is one set for each element in D), d_{def} (the default value of the signal).

Signal modes. The behaviour of a signal may differ between the different phases of operation of the system. Therefore, a signal can have different modes. A specific set of constraints is generated for each such mode, *i.e.* a signal with several modes has one parameter set P_{cont} or P_{disc} for each mode. The set used in a certain mode m is $P_{cont}(m)$ or $P_{disc}(m)$. Mode variables (m in this case) can be classified as discrete signals in themselves, so that error detection and recovery may also be implemented for them.

Modes may also be used to model certain dependencies between signals. That is, if the behaviour of signal A is limited due to the operational mode of signal B, these two signals can be grouped by means of signal modes representing this dependency. Furthermore, using different modes may increase the possibility of detecting errors.

An example of a signal that may have at least two different operational modes is *battery charge*. During normal operation it may be that the charge is only allowed to decrease or remain unchanged, whereas in a reloading mode the charge may only increase or remain unchanged. This would make error detection even more efficient than if the signal were modelled using only one mode allowing both behaviours.

2.2. Executable assertions and forced validity

Error detection is performed using the configuration parameters of the signals to build executable assertions. An error in a signal is detected as soon as the signal violates any of the constraints given by the configuration

Signal status	Test No.	Conditions for test	Validity constraint	Recovery
–	1	Always	$s \le s_{max}$	$s_r \leftarrow s_{max}$
	2	Always	$s \ge s_{min}$	$s_r \leftarrow s_{min}$
$s > s'$	3a	w = not allowed	$s - s' \le r_{max,incr}$	$s_r \leftarrow s' + r_{max,incr}$
	4a	$s - s' > r_{max,incr}$ and w = allowed	$(s' - s_{min}) + (s_{max} - s) \le r_{max,decr}$	$s_r \leftarrow (s_{max} - r_{max,decr}) + (s' - s_{min})$
	5a	$s - s' \le r_{max,incr}$ and ($s < s_{max}$ or w = allowed)	$s - s' \ge r_{min,incr}$	if $s_{max} - s' \ge r_{min,incr}$ then $s_r \leftarrow s' + r_{min,incr}$ else if w = allowed then $s_r \leftarrow (s_{min} + r_{min,incr}) - (s_{max} - s')$ else $s_r \leftarrow s_{max}$
$s < s'$	3b	w = not allowed	$s' - s \le r_{max,decr}$	$s_r \leftarrow s' - r_{max,decr}$
	4b	$s' - s > r_{max,decr}$ and w = allowed	$(s_{max} - s') + (s - s_{min}) \le r_{max,incr}$	$s_r \leftarrow (s_{min} + r_{max,incr}) - (s_{max} - s')$
	5b	$s' - s \le r_{max,incr}$ and ($s > s_{min}$ or w = allowed)	$s' - s \ge r_{min,decr}$	if $s' - s_{min} \ge r_{min,decr}$ then $s_r \leftarrow s' - r_{min,decr}$ else if w = allowed $s_r \leftarrow (s_{max} - r_{min,decr}) + (s' - s_{min})$ else $s_r \leftarrow s_{min}$
$s = s'$	3c	$r_{min,incr} = 0$ and $r_{max,incr} = 0$ and ($s > s_{min}$ or w = allowed)	$r_{min,decr} = 0$	if $s' - s_{min} \ge r_{min,decr}$ then $s_r \leftarrow s' - r_{min,decr}$ else if w = allowed $s_r \leftarrow (s_{max} - r_{min,decr}) + (s' - s_{min})$ else $s_r \leftarrow s_{min}$
	4c	$r_{min,decr} = 0$ and $r_{max,decr} = 0$ and ($s < s_{max}$ or w = allowed)	$r_{min,incr} = 0$	if $s_{max} - s' \ge r_{min,incr}$ then $s_r \leftarrow s' + r_{min,incr}$ else if w = allowed then $s_r \leftarrow (s_{min} + r_{min,incr}) - (s_{max} - s')$ else $s_r \leftarrow s_{max}$
	5c	not ($r_{min,decr} = 0$ and $r_{max,decr} = 0$) and not ($r_{min,incr} = 0$ and $r_{max,incr} = 0$) and ($s < s_{max}$ or w = allowed)	$r_{min,incr} = 0$ or $r_{min,decr} = 0$	if $s_{max} - s' \ge r_{min,incr}$ then $s_r \leftarrow s' + r_{min,incr}$ else if w = allowed then $s_r \leftarrow (s_{min} + r_{min,incr}) - (s_{max} - s')$ else $s_r \leftarrow s_{max}$
	5c'	not ($r_{min,decr} = 0$ and $r_{max,decr} = 0$) and not ($r_{min,incr} = 0$ and $r_{max,incr} = 0$) and ($s > s_{min}$ or w = allowed)	$r_{min,incr} = 0$ or $r_{min,decr} = 0$	if $s' - s_{min} \ge r_{min,decr}$ then $s_r \leftarrow s' - r_{min,decr}$ else if w = allowed $s_r \leftarrow (s_{max} - r_{min,decr}) + (s' - s_{min})$ else $s_r \leftarrow s_{min}$

Table 2. Error detection and recovery for continuous signals.

Signal class	Validity constraint	Recovery
Random	$s \in D$	$s_r \leftarrow d_{def}$
Linear sequential	$s \in D$	$s_r \leftarrow T(s')$
	$s \in T(s')$	
Non-linear sequential	$s \in D$	$s_r \leftarrow d_{def}$
	$s \in T(s')$	

Table 3. Error detection and recovery for discrete signals.

parameters. The recovery mechanism is a rather intuitive scheme called forced validity, which assigns a recovery value within the valid domain of the signal (either as close as possible to the erroneous value or a default value). The validity constraints and recovery mechanisms for continuous and discrete signals are shown in Tables 2 and 3, respectively. In these tables, s is the current signal value, s' is the previous signal value, s_r is the recovered signal, and the symbol ← denotes "is assigned". Note that error detection and recovery for continuous signals is only dependent on the parameters, not the signal class.

For continuous signals, there are different validity constraints depending on the relationship between s and s', as indicated by the *Signal status* column. Each set of tests is performed in the order given by the *Test No.* column. A validity constraint is used when the conditions in the *Conditions for test* column are true. If a constraint is violated, the corresponding recovery mechanism is used immediately, before any further tests are performed. This means that if for example a signal is above its maximum limit it will be set to s_{max} in test 1 and will therefore no longer be above its maximum for the remaining tests. The constraint in 5c is violated only if the signal is not allowed to remain unchanged (unless it already is at its extreme values and is not allowed to wrap around). Therefore, the recovery mechanism in test 5c, "always increase", is a matter of choice. One may also choose recovery according to 5c', "always decrease". In our case study, we implemented mechanism 5c.

For discrete signals, the validity constraints are always tested. If a constraint is violated, the corresponding recovery mechanism is used and the test is terminated.

Since the mechanisms for error detection and recovery are parameterised, it is possible to formally verify the algorithms used in the mechanism. This can totally eliminate the probability of faults in the mechanisms.

2.3. Location and parameters

A number of different methods may be used to determine where the assertions and recovery mechanisms should be placed. From system design, the software should already be divided into functional blocks. In safety-critical systems, FMECA (Failure Mode Effect and Criticality Analysis) is widely used as a method for identifying the safety critical parts of the system and assessing the consequences of failures in these parts.

Parameter information may be obtained by the characteristics of the system itself. For instance, sensors naturally have a time constant dictating the maximum rate of change for the data provided by that sensor. Properties of the physical surroundings of the systems are also a source of parameter values. For discrete signals, typical sources of information are allowed settings on user panels, or internal state machines.

The process of gathering information for parameter values for executable assertions forces developers to review the system they have developed. This may assist in

identifying contradicting specifications and/or parts that have not yet been properly analysed.

3. Case study

As an assessment of the effect of the proposed scheme on the failure symptoms of an embedded control system, we conducted an evaluation using error injection.

3.1. Target system

The target system is an aircraft-arresting system resembling those found on runways and aircraft carriers. The specifications of the system are based on specifications found in [10]. It consists of a cable strapped between two tape drums, one on each side of the runway (see Figure 4). Two computer nodes control the drums: one master node and one slave node. An incoming aircraft catches the cable by means of a hook, and a rotation sensor on the master drum periodically tells the master node the length of the pulled out cable. The master node calculates the set point pressure to be applied to the drums by means of hydraulic pressure valves. The pressure slows the rotation of the drums and brings the aircraft to a halt. The slave node receives its set point pressure value from the master node and applies this to its drum. Pressure sensors on the valves give feedback to their respective nodes about the pressure that is actually being applied so that a software-implemented PID-regulator can keep the actual pressure as close to the set point pressure as possible.

Physical limitations put constraints on the system:

- Retardation (r). The retardation of the aircraft shall not have a negative effect on the pilot. Constraint: $r < 2.8g$
- Retardation force (F_{ret}). The retarding force shall not exceed the structural limitations of the aircraft. Constraint: $F_{ret} < F_{max}$. The maximum allowed forces (F_{max}) are defined for several aircraft masses and engaging velocities in [10].
- Stopping distance (d). The braking distance of the aircraft shall not exceed the length of the runway. Constraint: $d < 335$ m

A violation of one or more of these constraints is defined as a *failure*.

3.2. Software instrumentation

The software of the system consists of a number of periodic processes and one main background process. An overview of the basic architecture is shown in Figure 3.

DIST_S monitors the rotation sensor and provides a total count of the pulses, *pulscnt*, generated during the arrestment. CLOCK provides a clock, *mscnt*, with one millisecond resolution. CALC (which is the main background process) uses those two signals to calculate a set point value for the pressure valves, *SetValue*, at predefined checkpoints along the runway. The number of the current checkpoint is stored in the checkpoint counter, *i*. The actual pressure applied by the valves, *IsValue*, is provided by PRES_S, which monitors the pressure sensor. V_REG uses the signals SetValue and IsValue to control *OutValue*, the output value to the pressure valve. PRES_A uses the OutValue signal to set the pressure valve. All modules are periodic except for CALC, which runs when the other modules are dormant. CLOCK and DIST_S both have a period of 1 ms and the other modules have periods of 8 ms.

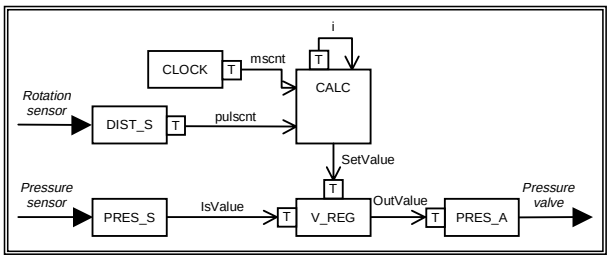

Figure 3. The basic software architecture.

The signals in Figure 3 are service critical, *i.e.* they are essential for the system to provide proper service and for this experiment were selected as test signals. The classifications and parameters of the signals are seen in Table 4.

Signal	Producer	Consumer	Test location	Class
i	CALC	CALC	CALC	CMD
SetValue	CALC	V_REG	V_REG	CR
OutValue	V_REG	PRES_A	PRES_A	CR
mscnt	CLOCK	CALC	CLOCK	CMS
pulscnt	DIST_S	CALC	DIST_S	CMD
IsValue	PRES_S	V_REG	V_REG	CR

Table 4. Classification of the signals in the target system.

In Table 4, the *Producer* is the originating module of a signal, the *Consumer* is the receiving module, and the *Test Location* is where the executable assertion and force validity mechanism were placed. The *Class* is how the signal was classified (C = continuous, R = random, M = monotonic, S = static rate, D = dynamic rate).

Using these classifications, we constructed executable assertions and recovery mechanisms as described in section 2. The locations of these assertions are shown in Figure 3 above (the small boxes with T's inside).

Note that no signals were classified as discrete. In this system there is only one signal which could have been classified as discrete and that is the checkpoint counter, *i*, since it dictates a sequence (the checkpoints must come in a certain order).

3.3. Fault injection environment

As seen in Figure 4, the target system was hooked up to the fault injection experiment system FIC^3 (Fault Injection Campaign Control Computer, see [11] for details).

The FIC^3 is capable of injecting errors into the target system. The injected errors consist of modifications of processor registers and memory areas where variables of the software modules are stored. Previous studies have shown that injecting bit-flips into a system using software-implemented fault-injection (SWIFI) closely resembles the behaviour of hardware failures [12]. An environment simulator acts as the barrier and as the incoming aircraft.

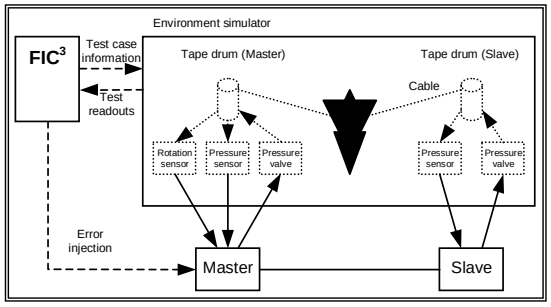

Figure 4. The FIC^3 and the target system.

3.4. Experimental set-up

We created two error sets: E_1 containing 160 errors, and E_2 containing 30 errors. Each error in E_1 is configured as a random bit-flip in either the stack area, the memory areas of the software modules, or the internal registers of the processor, modelling random hardware faults. The distribution of errors in E_1 is shown in Table 5.

Memory area	Size (bytes)	# errors (n_e)	Error numbers	# injections (n_e·25)
Stack	1008	25	1-25	625
CALC	164	55	26-80	1375
CLOCK	12	15	81-95	375
PRES_A	12	10	96-105	250
PRES_S	2	5	106-110	125
DIST_S	13	15	111-125	375
V_REG	36	15	126-140	375
Registers	3	20	141-160	500
Total	***1250***	***160***	–	***4000***

Table 5. The distribution of errors in E_1.

Signal	# errors (n_e)	Error numbers	# injections (n_e·25)
i	5	S1-S5	125
SetValue	5	S6-S10	125
OutValue	5	S11-S15	125
mscnt	5	S16-S20	125
pulscnt	5	S21-S25	125
IsValue	5	S26-S30	125
Total	***30***	–	***750***

Table 6. The distribution of errors in E_2.

The errors in E_2 specifically target the signals for which executable assertions and recovery mechanisms have been constructed (see Figure 3), and model errors in the monitored signals. The distribution of errors in E_2 is shown in Table 6.

All errors were injected in the master node. For each error in the error sets, the system was subjected to 25 test cases, *i.e.* incoming aircraft with velocity ranging from 40 m/s to 70 m/s, and mass ranging from 8000 kg to 20000 kg. The injections were made for the original system, *i.e.* without the executable assertions and recovery mechanisms, and for the instrumented system, *i.e.* with executable assertions and recovery mechanisms, giving a total of 9500 error injections.

The error sets were generated by random assignment of errors to the various memory areas and signals, but the same errors were used for both systems. The error injections were time triggered and were injected with a period of 10 ms (recall that most modules in the target system have a period of 8 ms). Thus, errors may have been injected during the execution of the executable assertions or the execution of the recovery mechanisms. The runs were recorded and compared.

4. Results

We injected errors in two embedded systems: the original target system and an instrumented version. Specifically, we injected two error sets, E_1 and E_2, containing 160 errors and 30 errors respectively, for 25 test cases in each system. Table 7 shows a summary of the results for E_1. Table 8 shows a summary of the results for E_2. For each area or signal, we see the number of injections and the number of resulting failures. The success rate is the normalised fraction of successful runs performed by the system. The number of failures and the success rate are shown for both systems. The reduction is a measure of how well the instrumented system handled failures as compared to the original system, and is calculated as $100\%\cdot(F_o\text{-}F_i)/F_o$. A 100%-reduction means that all failures in the original system were handled in the instrumented system. A negative reduction means that the number of failures increased in the instrumented system.

Table 9 shows the number of failed runs per test case for error set E_2. The *Mass* column contains the mass of the incoming aircraft and the *Velocity* column contains the engaging speed. All 30 errors in E_2 were injected for each test case. To determine the statistical significance of the difference between the original and the instrumented system we perform a t-test. Let the null hypothesis be: *the executable assertions and recovery mechanisms do not reduce the number of failures*. Data for error set E_2 (Table 9) give $t_0 = 1.7093$ with 48 degrees of freedom, and $P(t > t_0) = 0.047$. We can therefore reject the null hypothesis with $\alpha < 0.05$.

Memory area	# inj.	# fails (orig.) F_o	Success rate	# fails (instr.) F_i	Success rate	Reduction
Stack	625	52	0.9168	49	0.9216	5.77%
CALC	1375	57	0.9585	46	0.9665	19.30%
CLOCK	375	10	0.9733	21	0.9440	-110.10%
PRES_A	250	0	1.000	0	1.0000	N/A
PRES_S	125	3	0.9760	10	0.9200	-233.33%
DIST_S	375	65	0.8267	43	0.8853	33.85%
V_REG	375	5	0.9867	7	0.9813	-40.00%
Registers	500	277	0.4460	271	0.4580	2.17%
Total	***4000***	***469***	***0.8828***	***447***	***0.8883***	***4,69%***

Table 7. The results from the injection experiments with error set E_1.

Signal	# inj.	# fails (orig.) F_o	Success rate	# fails (instr.) F_i	Success rate	Reduction
i	125	48	0.6160	36	0.7120	25.00%
SetValue	125	4	0.9680	3	0.9760	25.00%
OutValue	125	1	0.9920	0	1.0000	100.00%
mscnt	125	9	0.9280	7	0.9440	22.22%
pulscnt	125	63	0.4960	39	0.6880	38.10%
IsValue	125	4	0.9680	2	0.9840	50.00%
Total	***750***	***129***	***0.8280***	***87***	***0.8883***	***32.56%***

Table 8. The results from the injection experiments with error set E_2.

Test case #	Mass (kg)	Velocity (m/s)	# fails (orig.)	#fails (instr.)	Diff.
1	8,000	40	1	1	0
2	8,000	48	1	1	0
3	8,000	55	4	2	-2
4	8,000	63	2	5	3
5	8,000	70	7	5	-2
6	11,000	40	1	0	-1
7	11,000	48	2	0	-2
8	11,000	55	2	0	-2
9	11,000	63	6	0	-6
10	11,000	70	8	4	-4
11	14,000	40	1	0	-1
12	14,000	48	2	1	-1
13	14,000	55	6	3	-3
14	14,000	63	6	6	0
15	14,000	70	9	6	-3
16	17,000	40	2	0	-2
17	17,000	48	6	1	-5
18	17,000	55	8	5	-3
19	17,000	63	8	7	-1
20	17,000	70	11	10	-1
21	20,000	40	1	0	-1
22	20,000	48	4	5	1
23	20,000	55	7	7	0
24	20,000	63	12	8	-4
25	20,000	70	12	10	-2
		Total	***129***	***87***	***-42***

Table 9. Number of failures per test case for error set E_2.

5. Discussion

We injected two sets of errors into our target system: one modelling random hardware faults (E_1), where each error was a bit-flip in random areas of the system memory and CPU registers, and one modelling data errors in specific signals (E_2), targeting the signals which were fitted with our mechanisms.

The errors were injected periodically with a period of 10 milliseconds. The target system has a main period of 8 milliseconds. Therefore, the errors are likely to affect the system in a manner that cannot be said to model software faults, since such faults would most likely induce data errors with a period matching that of the system.

The results from error set E_1 show that errors injected into the stack and the registers caused approximately the same amount of failures in both systems. It may be argued that these errors are more malicious than those injected into the memory area of specific software modules and would very likely lead to control flow errors. The proposed mechanisms are not aimed at detecting or recovering from such errors.

For the different software modules, the reduction in failures induced by random hardware faults varies (error set E_1, see Table 7). The low overall reduction for errors injected randomly in the memory areas of the modules is mainly due to the following reasons:

1. Errors occurred with a period not matching that of the system. This increased the probability of errors occurring between the test of a signal and the usage of that signal, thereby nullifying the effect of any recovery that may have been performed.
2. Errors were injected into variables not covered by assertions and recovery mechanisms. These errors are likely to affect the system in a way that the executable assertions cannot detect.
3. Errors were injected into variables belonging to the executable assertions and recovery mechanisms. Since these mechanisms were inactive in the original system, those errors did not cause any failures, whereas in the instrumented system, where the executable assertions and recovery mechanisms were active, they caused failures.

Point 1 highlights a fundamental difference between software-based and hardware-based fault-tolerance techniques. Whereas the hardware-based techniques are always active and ready to handle errors, the software-based techniques are active only at certain points in time. If a data error occurs between the execution of a test on the data and the usage that data, the software-based techniques cannot detect the error, much less recover from it.

Point 2 shows that error detection and recovery mechanisms aimed at specific signals and data areas are not effective against errors occurring in signals related to the monitored ones.

Point 3 shows that it is very important to separate the memory areas of error detection and recovery mechanisms from the memory areas of the application. Preferably, the mechanisms should also be located in other, more reliable, memory circuits.

The results from error set E_2 (Table 8) show that when the errors directly affect variables monitored by the mechanisms, the reduction in the number of resulting failures is greater than if the errors affect randomly chosen

variables. These findings suggest that the locations of the detection and recovery mechanisms largely influence the degree of their success, which is consistent with findings reported in [7]. Our results indicate that error detection and recovery mechanisms should be located as close to the receiver of a signal as possible or be performed when the receiver of a signal accesses the value.

The system used in the case study did not contain any signals we could classify as discrete signals, making the evaluation valid only for continuous signals. This, however, may indicate that the majority of signals in an embedded system may be modelled as continuous signals.

6. Summary and future work

In this paper, we have proposed a classification method for internal signals in embedded control systems, enabling the construction of parameterised error detection and error recovery mechanisms. For error detection, we set up several formal constraints that have to be fulfilled by the signals and that are tested in executable assertions. An error is detected if any of these constraints is violated. The proposed error recovery mechanism forces an erroneous signal into its valid domain by assigning it a "best effort" value according to the parameters determined by the classification of the signal (forced validity).

A case study was performed to study the effect on the failure behaviour of an embedded system. Two versions of an aircraft arresting system – one without and one with the proposed mechanisms – were compared. Both versions were subjected to two error sets that modelled intermittent hardware faults: one exercising random bit-flips in memory areas and processor registers, and one specifically targeting the monitored signals.

The results show that the location of executable assertions and recovery mechanisms is of the utmost importance to the effectiveness of the mechanisms. Errors that occur in data not monitored by the error detection and recovery mechanisms are poorly handled. Also, the memory areas of the mechanisms should be separated from the application memory or else the mechanisms will be as vulnerable to errors as the variable they monitor.

Failures induced by random errors in stack and CPU registers were not significantly affected. These errors more likely lead to control flow errors, which the proposed mechanisms cannot detect or recover from.

This study has not examined the capability of the proposed error detection and recovery mechanisms on a detailed level. Further studies will include a study of the error detection coverage given that an error does exist in the tested signal and the error recovery coverage given that an error is detected. We will also study errors induced by software faults using fault injection experiments, *i.e.* altering the source code to emulate software faults.

Acknowledgement

We would like to thank Robert Feldt, Marcus Rimén and Jörgen Christmansson for their comments on this paper, and Christine Räisänen for language support. We are also grateful for the comments of the reviewers, which helped to increase the quality of this paper. This research was financially supported by Volvo and by the National Board for Industrial and Technical Development (NUTEK), Sweden, under contract 1P21-97-4745.

References

[1] Avizienis A., "The N-Version Approach to Software Fault-Tolerance", *IEEE Transactions on Software Engineering*, Vol. 11, No 12, pp. 1491-1501, 1985

[2] Randell B., "System Structure for Software Fault-Tolerance", *IEEE Transactions on Software Engineering*, Vol. 1, No. 2, pp. 220-232, 1975

[3] Randell B., Xu J., "The evolution of the recovery block concept", *Software Fault Tolerance*, Lye M.R. (ed.), Chapter 1, Willey, 1995

[4] Andrews D.M., "Using Executable Assertions for Testing and Fault Tolerance", *Proceedings 9th International Symposium on Fault-Tolerant Computing*, pp. 102-105,1979

[5] Mahmood A., Andrews D.M., McCluskey E.J., "Executable Assertions and Flight Software", *Proceedings 6th Digital Avionics Systems Conference*, pp. 346-351, Baltimore (MD), USA, AIAA/IEEE, 1984

[6] Rabéjac C., Blanquart J.-P., Queille J.-P., "Executable Assertions and Timed Traces for On-Line Software Error Detection", *Proceedings 26th International Symposium on Fault-Tolerant Computing*, pp.138-147, 1996

[7] Leveson N.G., Cha S.S., Knight J.C., Shimeall T.J., "The Use of Self Checks and Voting in Software Error Detection: An Empirical Study", *IEEE Transactions on Software Engineering*, Vol. 16, No. 4, pp. 432-443, 1990

[8] Stroph R., Clarke T., "Dynamic Acceptance Tests for Complex Controllers", *Proceedings 24th Euromicro Conference*, pp.411-417, 1998

[9] Clegg M., Marzullo K., "Predicting Physical Processes in the Presence of Faulty Sensor Readings", *Proceedings 27th International Symposium on Fault-Tolerant Computing*, pp.373-378, 1996

[10] US Air Force – 99, "Military specification: Aircraft Arresting System BAK-12A/E32A; Portable, Rotary Friction", *MIL-A-38202C*, Notice 1, US Department of Defence, September 2, 1986

[11] Christmansson J., Hiller M., Rimén M., "An Experimental Comparison of Fault and Error Injection", *Proceedings 9th International Symposium on Software Reliability Engineering*, pp. 369-378, 1998

[12] Rimén M., Ohlsson J., Torin J., "On Microprocessor Error Behavior Modelling", *Proceedings 24th International Symposium on Fault-Tolerant Computing*, pp.76-85, 1994