Executable Assertions for Detecting Data Errors in Embedded Control Systems[*]

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

Abstract

In order to be able to tolerate the effects of faults, we must first detect the symptoms of faults, i.e. the errors. This paper evaluates the error detection properties of an error detection scheme based on the concept of executable assertions aiming to detect data errors in internal signals. The mechanisms are evaluated using error injection experiments in an embedded control system. The results show that using the mechanisms allows one to obtain a fairly high detection probability for errors in the areas monitored by the mechanisms. The overall detection probability for errors injected to the monitored signals was 74%, and if only errors causing failure are taken into account we have a detection probability of over 99%. When subjecting the target system to random error injections in the memory areas of the application, i.e., not only the monitored signals, the detection probability for errors that cause failure was 81%.

Keywords: signal classification scheme, executable assertions, error detection, software implemented fault tolerance, fault injection

1. Introduction

Fault-tolerance is no longer required only in high-end systems such as aircraft, nuclear power plants or spacecraft. Consumer products, such as automobiles, are increasingly dependent on electronics and software and require low-cost techniques for achieving fault-tolerance. Low-cost in this sense means that these techniques are inexpensive to develop and that the product is (relatively) inexpensive to produce.

The first step in tolerating the effects of faults is to detect the symptoms of faults, i.e. the errors. Several techniques and methods have been proposed for error detection. An NVP-style approach to error detection is achieved by running several versions or variants of the system in parallel and then compare their results [1]. If the results differ, an error must have occurred in at least one of the versions. This approach is very effective but tends to be also very expensive. A more inexpensive way of error detection is to explicitly check for errors in the system-state. Several techniques for such self-tests have been proposed (e.g. [2][3][4]), but in many cases little is known about their effectiveness.

Most self-tests are based on the concept of executable assertions [5][6]. Executable assertions are commonly statements, which can be made about the variables in a program. These statements are executed in on-line tests to see if they hold true. If they do not, an error has occurred and processes for assessment and recovery may be invoked. In addition to on-line error detection, executable assertions may be used during the development of a system for testing purposes [7] and to assess the vulnerability of the system.

Self-tests, as for instance executable assertions, also play major roles in software fault tolerance structures such as Recovery Blocks (RB) [8] and its variants (e.g. Consensus RB [9] and Distributed RB [10]), and other structures (e.g. N Self-Checking Components [11], N Copy Programming or Retry Blocks [12]).

The effectiveness of executable assertions is highly application dependant. In order to develop tests with high error detection coverage, the developers require extensive knowledge of the system. Introducing rigorous ways of defining the statements used for executable assertions, or even better, providing generic mechanisms that can be instantiated by parameters alone, reduce the importance of this drawback.

This paper evaluates the detection capabilities of error

[*] This research was supported by Volvo and by the National Board for Industrial and Technical Development (NUTEK), Sweden, under contract 1P21-97-4745.

detection mechanisms based on the executable assertion concept, that work on a signal-basis, meaning that only one signal/variable is tested in each individual test routine. The paper also proposes a defined process for incorporating the mechanisms into a system.

In order to evaluate the error detection capabilities of the proposed mechanisms we performed a case study using error injection experiments on an embedded system used for arresting aircraft on a runway. The aim of this study was to investigate the probability of detecting erroneous states induced by internal data errors. We also measured the detection latency as being the time from the first injection of an error to the first detection. Even though the error detection mechanisms may detect errors induced by software faults as well as hardware faults, the case study concentrates on errors induced by hardware faults.

The results show that given that an error is present in a monitored signal, and that this error leads to system failure, the detection probability is over 99%. For error injections into random locations in the memory areas of the target system, the errors that caused system failure were detected with a probability of over 81%. The presented technique is therefore a viable candidate for error detection with reasonably high detection coverage if costs have to be kept low.

Section 2 contains a description of the error detection scheme used for this evaluation. Section 3 describes the case study and 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.

2. Executable assertions

Error detection in the form of executable assertions can potentially detect any error in internal data caused by either software faults or hardware faults [13]. 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

One of the main drawbacks of executable assertions, and indeed of all kinds of acceptance tests, is that they are very application specific. One way of lessening the impact of this specificity is to devise a rigorous way of classifying the data that are to be tested. A classification scheme will help when determining the valid domain for the signals. The classification scheme used in this investigation is shown in Figure 1. Below is a description of the classification scheme.

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

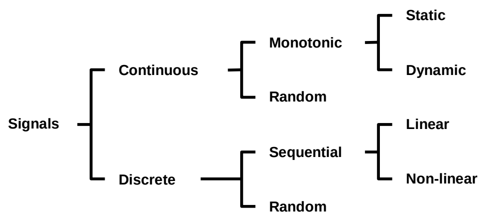

Figure 1. Signal classification scheme.

Error detection is performed as a test of the constraints. A violation of a constraint is interpreted as the detection of an error.

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.

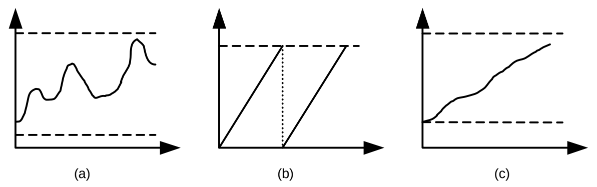

Figure 2. Continuous signals: (a) random, (b) static monotonic (with wrap-around), (c) dynamic monotonic

For the proposed error detection and recovery mechanisms, 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.

Table 1. Parameter constraints for continuous signal classes.

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	$r_{\max,incr} \geq r_{\min,incr} \geq 0$, $r_{\max,decr} \geq r_{\min,decr} \geq 0$

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 [4] and [14] 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 mechanisms we assign to each signal a set P_{disc} containing the following parameters: D (the set of valid values) and $T(d)$ (the set of valid transitions from element d in D; there is one set for each element in D).

A typical example of a discrete signal is a state variable. From any given state, the variable may be set to a (fixed) number of other states. Consider for example the state diagram shown in Figure 3. There are five states ($v1$ through $v5$) and a number of transitions between these states. The valid domain is therefore $D = \{v1, v2, v3, v4, v5\}$ and the transition sets are $T(v1) = \{v2, v4\}$, $T(v2) = \{v3, v4\}$, $T(v3) = \{v4\}$, $T(v4) = \{v5\}$, and $T(v5) = \{v1\}$.

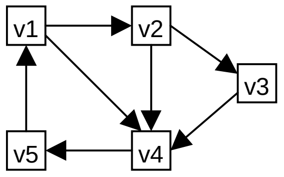

Figure 3. Example state diagram for a non-linear sequential discrete signal.

Signal modes. The behaviour of a signal may differ between 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 may be implemented for them as well.

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.

2.2. Error detection

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 the constraints given by the configuration parameters. The executable assertions for continuous and discrete signals are shown in Tables 2 and 3, respectively. In these tables, s is the current signal value and s' is the previous signal value.

Table 2. Executable assertions for continuous signals

Signal status	Test No.	Assertion	Description
-	1	$s \leq s_{\max}$	Maximum value
	2	$s \geq s_{\min}$	Minimum value
$s > s'$	3a	$s - s' \leq r_{\max,incr} \wedge$ $s - s' \geq r_{\min,incr}$	Within increase parameters
	4a	w = allowed $\wedge$ $(s' - s_{\min}) + (s_{\max} - s) \leq r_{\max,decr} \wedge$ $(s' - s_{\min}) + (s_{\max} - s) \geq r_{\min,decr}$	Wrap-around is allowed and within decrease parameters
$s < s'$	3b	$s' - s \leq r_{\max,decr} \wedge$ $s' - s \geq r_{\min,decr}$	Within decrease parameters
	4b	w = allowed $\wedge$ $(s_{\max} - s') + (s - s_{\min}) \leq r_{\max,incr} \wedge$ $(s_{\max} - s') + (s - s_{\min}) \geq r_{\min,incr}$	Wrap-around is allowed and within increase parameters
$s = s'$	3c	$r_{\min,incr} = 0 \wedge$ $r_{\max,incr} = 0 \wedge$ $r_{\min,decr} = 0$	Monotonically decreasing signal and within decrease parameters
	4c	$r_{\min,decr} = 0 \wedge$ $r_{\max,decr} = 0 \wedge$ $r_{\min,incr} = 0$	Monotonically increasing signal and within increase parameters
	5c	$\neg (r_{\min,decr} = 0 \wedge r_{\max,decr} = 0) \wedge$ $\neg (r_{\min,incr} = 0 \wedge r_{\max,incr} = 0) \wedge$ $(r_{\min,incr} = 0 \vee r_{\min,decr} = 0)$	Random signal and within parameters

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. The *Assertion* column contains the assertions that the signals must pass. In the *Description* column is a short description of the implications of passing a particular test.

Each time a signal is tested, it is subjected to at most five assertions. The first two tests, Test No. 1 and 2, are always used, regardless of the signal status, whereas the remaining tests are chosen depending on the relationship between the previous signal value and the current signal value. If either of the first two tests fails, the entire test fails. However, if the first two tests are passed, only one of the remaining assertions must be fulfilled.

Table 3. Executable assertions for discrete signals

Signal class	Assertion	Comment
Random	$s \in D$	
Sequential	$s \in D$	
	$s \in T(s')$	*This property actually implies that $s \in D$, but both tests are used nonetheless.*

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

Since the mechanisms for error detection are general algorithms that are instantiated with parameters, it is possible to formally verify the algorithms, which can totally eliminate the probability of faults in them, although faulty parameters may still be a problem. However, the parameters may be calibrated using fault injection experiments.

2.3. Location and parameters

A number of different methods may be used to determine which signals should be monitored and where the executable assertions 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. The following is our proposed process for equipping a system with error detection mechanisms as described in this paper:

1. Identify the input and output signals of the system.
2. Identify the signal pathways from each input signal through the system and to one or more output signals.
3. Identify internally generated signals that have a direct influence on intermediate and output signals.
4. Determine which of the identified signals are the most crucial for flawless operation of the system and should therefore be monitored by error detection mechanisms, e.g. by using FMECA.
5. Classify each signal found in (4) according to the scheme described above.
6. Determine values for the characterising parameters of the signals. Remember that a signal may behave differently for different modes of operation in the system.
7. Decide on locations for the mechanisms.
8. Incorporate the mechanisms in the system.

2.4. Error detection coverage

The detection coverage that may be obtained with these mechanisms is very dependent on the characteristics of the errors that may occur. If we, given that an error has occurred, define the probabilities P_{em} = Pr{error location is in a monitored signal}, P_{en} = Pr{error location is *not* in a monitored signal} = 1 - P_{em}, P_{prop} = Pr{error propagates to a monitored signal}, and P_{ds} = Pr{an error is detected given that the error is located in a monitored signal}. The total probability of detecting an error that is present can than be written as $P_{detect} = (P_{en}P_{prop} + P_{em})P_{ds}$. For a given system, the probability P_{ds} can be assessed separately from the other probabilities and is independent of the probability distributions for error occurrence and error location. A common way of performing such an assessment is by conducting error injection experiments. We have performed a case study to assess P_{detect} and P_{ds} for a given target system (see the following sections).

3. Case study

As an assessment of the effectiveness of the error detection mechanisms when employed in 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 purpose of this system is to assist incoming aircraft in reducing their velocity to a complete halt. The specifications of the system are based on specifications found in [15]. Our experiments were performed on an actual implementation of this system, i.e. no simulations (other than environment simulations) were used.

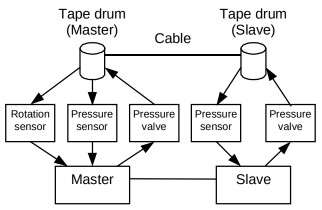

Figure 4. The experiment target: an aircraft arresting system.

System overview. The system 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 as possible.

Software overview. The software of the master node of the system consists of a number of periodic processes and one main background process. An overview of the basic software architecture can be seen in Figure 5.

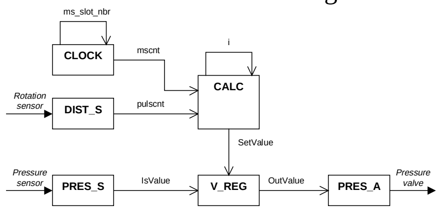

Figure 5. The basic software architecture.

CLOCK provides a clock, *mscnt*, with one millisecond resolution. The signal *ms_slot_nbr* tells the module scheduler (which is a part of the CLOCK module) which the current slot is. The system operates in seven 1-ms-slots. In each slot, one or more of the other modules (except for CALC) are invoked.

DIST_S monitors the rotation sensor and provides a total count of the pulses, *pulscnt*, generated during the arrestment. The rotation sensor reads the number of pulses generated by a tooth wheel on the tape drum.

CALC (which is the main background process) uses the signals *mscnt* and *pulscnt* to calculate a set point value for the pressure valves, *SetValue*, at six predefined checkpoints along the runway. The distance between these checkpoints is constant, and they are detected by comparing the current *pulscnt* with internally stored *pulscnt*-values corresponding to the various checkpoints. The number of the current checkpoint is stored in the checkpoint counter, *i*.

PRES_S monitors the pressure sensor measuring the pressure that is actually being applied by the pressure valves. This value is provided in the signal *IsValue*.

V_REG uses the signals *SetValue* and *IsValue* to control *OutValue*, the output value to the pressure valve. *OutValue* is based on *SetValue* and then modified to compensate for the difference between *SetValue* and *IsValue*. This module contains the software implemented PID-regulator.

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, i.e., it runs in the background. CLOCK and DIST_S both have a period of 1 ms and the other modules have periods of 7 ms.

The software of the slave node is slightly different from that of the master node. No calculations of set point values for the applied pressure are performed. The slave node simply receives a set point value from the master node, which it then applies to its tape drum. The modules existing also in the slave node are PRES_S, V_REG, CLOCK, and PRES_A. The modules DIST_S and CALC are not present.

Failure classification. The specifications from which the system is implemented [15] clearly dictate certain physical constraints, which the system must honour. These constraints are that the retardation must not exceed a certain limit in order to not affect either the plane or the pilot in a negative way, and that the force applied to the aircraft by the cable must not exceed certain limits in order to not endangering the aircraft. Also, the length of the runway is limited. However, this constraint may vary from instalment to instalment. The constraints are as follows:

1. Retardation (*r*). The retardation of the aircraft shall not have a negative effect on the pilot. Constraint: $r < 2.8g$
2. 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 [15]. Force constraints for combinations of masses and velocities other than those given in [15] are obtained using interpolation and extrapolation.
3. 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. This is a pessimistic failure classification, in the sense that not all arrestments which according to this classification were failures would have turned out to be critical in reality. For instance, in most cases a retardation of up to 3*g* will not significantly damage the aircraft or injure the pilot. The duration of a typical, failure-free, arrestment ranges from about 5 seconds (low kinetic energy) up to about 15 seconds (high kinetic energy).

3.2. Software instrumentation

Using the process described in section 2.3, we identified 7 signals (of a total of 24 signals) in the target system that are service critical, i.e. essential for providing proper service. The signals are shown in Figure 5. The classifications of the signals are seen in Table 4.

Table 4. Classification of the signals.

Signal	Producer	Consumer	Test location	Class
SetValue	CALC	V_REG	V_REG	Co/Ra
IsValue	PRES_S	V_REG	V_REG	Co/Ra
i	CALC	CALC	CALC	Co/Mo/Dy
pulscnt	DIST_S	CALC	DIST_S	Co/Mo/Dy
ms_slot_nbr	CLOCK	CLOCK	CLOCK	Di/Se/Li
mscnt	CLOCK	CALC	CLOCK	Co/Mo/St
OutValue	V_REG	PRES_A	PRES_A	Co/Ra

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 assertions were placed. The *Class* is how the signal was classified (Co = continuous, Ra = random, Mo = monotonic, St = static rate, Dy = dynamic rate, Di = discrete, Se = sequential, Li = linear).

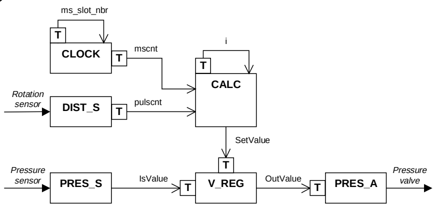

Figure 6. The locations of the executable assertions

Using these classifications, we constructed executable assertions as described in section 2. The locations of these assertions are shown in Figure 6 above (the small boxes with T's inside).

3.3. Fault injection environment

As seen in Figure 7, the target system was hooked up to the fault injection experiment system FIC^3 (Fault Injection Campaign Control Computer, see [16] for details).

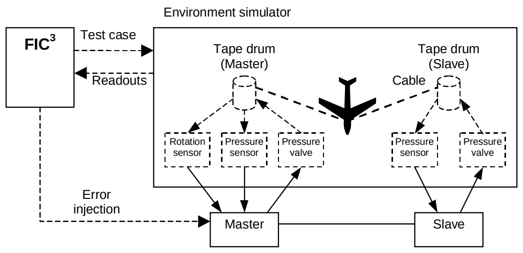

Figure 7. The FIC3 and the target system.

The FIC^3 is capable of injecting errors into the target system by means of SWIFI (SoftWare Implemented Fault Injection). Specifically, before initiating an experiment run, the FIC^3 downloads error parameters to an injection interrupt routine in the target system, which is then, during the experiment run, triggered by the FIC^3 when the actual injection is to be performed. The error detection mechanisms report detection by setting a digital output pin on the target processor high. This is detected by the FIC^3, which records and time-stamps the event. The injected errors consist of modifications of the memory areas where variables and signal values are stored. Previous studies have shown that injecting bit-flips into a system using SWIFI closely resembles the behaviour of hardware failures [17]. The downloaded injection parameters for this type of error are the address and bit position.

An environment simulator acts as the barrier (i.e. cable and tape drums) and as the incoming aircraft. This simulator is initialised using test case data (mass and incoming velocity). The FIC^3 triggers the simulator to start simulating an incoming aircraft. The simulator then feeds the system with sensory data (rotation sensor and pressure sensor) and receives actuator data (pressure value) from the system used for calculating new sensory data. All input to and output from the environment simulator is stored as experiment readouts and is subsequently analysed for system failure.

3.4. Experimental set-up

The experimental set-up calls for two error sets for evaluation purposes. In order to assess the probability P_{ds}, as defined in section 2.4, an error set $\boldsymbol{E_1}$ containing 112 errors was created. Each error in $\boldsymbol{E_1}$ is configured as a bit-flip in the monitored signals. Bit-flips can be used to model intermittent hardware faults, and it may be argued that using bit-flips in variables only may also model other faults inducing data errors in variables. Since single-bit errors are uniformly probable in all bit positions we chose to inject errors in each bit position of each signal in order to get a good estimate of the detection probability. Each signal is 16 bits long, hence, we have $7 \cdot 16 = 112$ errors in the error set. The distribution of errors in the error set is shown in Table 6.

Table 6. The distribution of errors in the error set $\boldsymbol{E_1}$.

Signal	Executable assertion	# errors (n_s)	Error numbers	# injections ($n_s \cdot 25$)
SetValue	EA1	16	S1-S16	400
IsValue	EA2	16	S17-S32	400
i	EA3	16	S33-S48	400
pulscnt	EA4	16	S49-S64	400
ms_slot_nbr	EA5	16	S65-S80	400
mscnt	EA6	16	S81-S96	400
OutValue	EA7	16	S97-S112	400
Total	–	***112***	–	***2800***

The other error set, $\boldsymbol{E_2}$, contains 200 errors configured as bit-flips in random bit positions in random locations (addresses) in application RAM (417 bytes) and stack (1008 bytes) areas, and is used to assess the total detection probability P_{detect} as described in section 2.4. These errors were selected from a uniform distribution (both location and bit-position), and the sampling was performed with replacement. Of the 200 errors, 150 were located in application RAM areas and 50 in the stack area.

All errors were injected in the master node. For each error in the error set, the system was subjected to 25 test cases, i.e. incoming aircraft, with velocity ranging uniformly from 40 m/s to 70 m/s, and mass ranging uniformly from 8000 kg to 20000 kg. For $\boldsymbol{E_1}$ we have $112 \cdot 25 = 2800$ different combinations $\langle m, v, e \rangle$ of mass, velocity and error and for $\boldsymbol{E_2}$ we have $200 \cdot 25 = 5000$ combinations. All test cases are such that if they are run on the target system without error injection, none of the error detection mechanisms report detection.

For $\boldsymbol{E_1}$, eight different versions of the system were tested – one for each of the seven individual executable assertions and one in which all seven executable assertions were active simultaneously. For each system every combination of mass, velocity and error was exercised, giving us a total of $2800 \cdot 8 = 22400$ experiment runs with error injections for $\boldsymbol{E_1}$. The error set $\boldsymbol{E_2}$ was used only on the version containing all seven executable assertions. Therefore we have 5000 experiment runs with error injections for $\boldsymbol{E_2}$.

The error injections were time triggered and were injected with a period of 20 ms (recall that most modules in the target system have a period of 7 ms). Thus, errors may have been injected during the execution of the executable assertions.

We say that we have successful error detection if an error is detected at least once during the entire observation period (40 seconds). The detection probability is then the probability of detecting an error at least once during the observation period. The detection latency is the time from the first injection of an error to the first reported detection.

4. Results

In Table 7, we can see the estimates of the detection probabilities per signal, per executable assertions and totals, as obtained using error set $\boldsymbol{E_1}$. The measures are calculated according to the formulas for coverage estimation in [18]. The measure $P(d) = n_d/n_e$ (where n_d is the number of runs in which errors were detected and n_e is the number of runs in which errors were injected) is an estimate of the probability that the error is detected during the observation time, $P(d|\text{fail}) = n_{d,\text{fail}}/n_{e,\text{fail}}$ (where we only take into account those runs in which the system failed) is an estimate of the probability that the error is detected given that a failure occurred, and $P(d|\text{no fail}) = n_{d,\text{no fail}}/n_{e,\text{no fail}}$ (where we only take into account those runs in which the system did not fail) is an estimate of the detection probability given that no failure occurred. The relation $n = n_{\text{fail}} + n_{\text{no fail}}$ holds for both errors and detections. For the individual signals we have $n_e = 400$ and for the totals we have $n_e = 2800$. The *All* column contains the results obtained when using the version of the software, which had all seven executable assertions activated simultaneously. The table also contains the 95% confidence intervals for the estimates of the detection probabilities. We can use the measure P(d) as an estimate of P_{ds} in the expression of the total detection probability for the entire system (see section 2.4). If a cell is empty in the table, this means that no detection was registered for that combination of signal and executable assertion.

The values shown in boldface are those that correspond to the "correct" signal-mechanism pair. For instance, the signal *SetValue* is directly monitored by mechanism EA1, and the signal *IsValue* is directly monitored by EA2

Table 7. Error detection probabilities (%) with confidence intervals at 95%. No confidence interval can be estimated for measured detection probabilities of 100.0%.

Signal	Measure	EA1	EA2	EA3	EA4	EA5	EA6	EA7	All
SetValue	P(d)	**55.5±4.1**	31.3±3.8	4.0±1.6				44.3±4.1	*59.5±4.0*
	P(d|fail)	**92.6±3.7**	72.4±6.4	1.5±1.7				87.9±4.7	*97.1±2.4*
	P(d|no fail)	**36.6±4.9**	10.5±3.1	5.3±2.3				22.8±4.2	*39.7±5.0*
IsValue	P(d)		**52.5±4.1**					47.0±4.1	*54.4±4.1*
	P(d|fail)		**89.6±7.3**					93.3±6.2	*100.0*
	P(d|no fail)		**47.4±4.4**					41.1±4.3	*47.2±4.4*
i	P(d)	26.8±3.6	29.8±3.8	**100.0**	1.5±1.0	1.0±0.8	0.5±0.6	47.8±4.1	*100.0*
	P(d|fail)	33.7±7.8	55.4±8.2	**100.0**	2.0±2.3	2.3±2.1	1.1±1.8	78.0±6.8	*100.0*
	P(d|no fail)	24.4±4.1	21.1±3.9	**100.0**	1.3±1.1	0.4±0.6	0.3±0.5	37.7±4.6	*100.0*
pulscnt	P(d)	50.3±4.1	42.8±4.1	0.3±0.4	**12.8±2.7**			0.3±0.4	*100.0*
	P(d|fail)	38.1±5.3	34.5±4.8	0.3±0.5	**0.0**			0.7±1.2	*100.0*
	P(d|no fail)	66.9±6.0	58.3±6.9	0.0	**16.6±3.5**			0.0	*100.0*
ms_slot_nbr	P(d)		20.0±3.3			**100.0**		6.8±2.1	*100.0*
	P(d|fail)		34.6±5.7			**100.0**		11.6±3.9	*100.0*
	P(d|no fail)		7.1±2.9			**100.0**		2.7±1.8	*100.0*
mscnt	P(d)	8.3±2.3	12.3±2.7				**100.0**	17.5±3.1	*100.0*
	P(d|fail)	20.0±13.4	18.2±13.8				**100.0**	13.0±11.8	*100.0*
	P(d|no fail)	7.5±2.2	11.9±2.7				**100.0**	17.8±3.2	*100.0*
OutValue	P(d)		1.0±0.8					**11.3±2.6**	*4.0±1.6*
	P(d|fail)		33.3±34.7					**85.7±23.5**	*100.0*
	P(d|no fail)		0.5±0.6					**9.9±2.5**	*3.3±1.5*
Total	P(d)	*20.1±1.2*	*27.1±1.4*	*14.9±1.1*	*2.0±0.4*	*14.4±1.1*	*14.4±1.1*	*25.0±1.3*	***74.0±1.4***
	P(d|fail)	*35.0±2.9*	*47.0±3.0*	*12.2±1.9*	*0.3±0.4*	*21.7±2.3*	*3.2±1.0*	*42.7±3.3*	***99.6±0.3***
	P(d|no fail)	*14.9±1.3*	*19.7±1.4*	*16.0±1.4*	*2.5±0.5*	*11.1±1.2*	*19.0±1.5*	*19.9±1.4*	***60.6±1.9***

Table 8. Error detection latencies for all errors (milliseconds).

Signal	Latency	EA1	EA2	EA3	EA4	EA5	EA6	EA7	All
SetValue	Min	**160**	570	50				20	*20*
	Average	**690**	2445	1241				842	*692*
	Max	**6259**	5588	6099				5297	*6490*
IsValue	Min		**10**					10	*20*
	Average		**612**					654	*1046*
	Max		**8142**					4466	*6630*
i	Min	311	270	**80**	2584	4686	3495	151	*100*
	Average	2125	2100	**210**	4381	5538	3891	1900	*228*
	Max	11397	8272	**401**	5798	7601	4286	6499	*421*
pulscnt	Min	390	1182	1563	**20**			230	*20*
	Average	1371	1379	1563	**239**			230	*272*
	Max	2284	2283	1563	**921**			230	*1803*
ms_slot_nbr	Min		1172			**20**		1703	*20*
	Average		3654			**32**		3462	*32*
	Max		8912			**140**		5738	*80*
mscnt	Min	1112	1352				**10**	1091	*20*
	Average	2050	1741				**25**	1673	*23*
	Max	4196	3525				**60**	3415	*61*
OutValue	Min		440					**20**	*2413*
	Average		1344					**1604**	*3379*
	Max		2704					**6179**	*7781*
Total	*Min*	*160*	*10*	*50*	*20*	*20*	*10*	*10*	***20***
	Average	*1286*	*1725*	*248*	*727*	*126*	*163*	*1314*	***511***
	Max	*11379*	*8912*	*6099*	*5798*	*7601*	*4286*	*6499*	***7781***

Table 9. Results for error set E_2

Area	Detection probability (%, 95% conf. int.)		Detection latency (ms, totals)		Detection latency (ms, failures)	
RAM	P(d)	12.8±0.9	Min	20	Min	20
	P(d|fail)	81.1±6.8	Average	1359	Average	1203
	P(d|no fail)	11.1±0.9	Max	5608	Max	5608
Stack	P(d)	4.2±0.9	Min	20	Min	20
	P(d|fail)	13.7±4.7	Average	250	Average	2077
	P(d|no fail)	2.9±0.8	Max	2684	Max	6449
Total	P(d)	10.6±0.7	Min	20	Min	20
	P(d|fail)	39.4±5.2	Average	1086	Average	1298
	P(d|no fail)	9.2±0.7	Max	5608	Max	6449

In Table 8 are the detection latencies measured during our experiments. The value is the time from the first injection of an error until the first registered detection, and it is measured in milliseconds. The table contains the minimum, average and maximum values for the detection latencies. Again, the boldface values correspond to the primary signal-mechanism pairs. In this table we consider all detected errors, those leading to failure as well as those not leading to failure.

The results from the experiments with error set E_2 are shown in Table 9. The table contains detection coverage with 95% confidence intervals and detection latencies measured in milliseconds. As with the measures for error set E_1, we used the formulas described in [18] to derive the probabilities shown in the table. The probabilities shown in Table 9 are estimates of P_{detect}, whereas the probabilities shown in Table 7 are estimates of P_{ds} (for more information on the definition of these probabilities, see section 2.4).

5. Discussion

The results obtained in this evaluation are specific for the target system, the error model and the test cases we have chosen. For other systems, error models, and/or test cases the results may vary. Having said that, we can now start our discussion of the results shown in the previous section.

5.1. Error detection probability, P_{ds}

This section discusses the results obtained with error set E_1. The results are the estimated values for the probability P_{ds}, i.e. the probability that an error is detected given that an error is present in one of the monitored signals and therefore can be detected by the mechanisms.

The overall detection probability was 74%, and if we consider the errors that lead to failure, as defined in section 3, the detection probability was over 99%. Roughly, 60% of the errors that did not lead to failure were detected. If we examine the individual executable assertions, we have detection probabilities ranging from just over 11% up to 100%.

The assertions that achieved a 100% detection probability monitored signals that were all essentially counters by nature; they were periodically incremented by some limited (small) amount. This makes errors easy to detect since the freedom of change was very small in these signals. We must remember that it is possible, even probable, that we do not achieve a 100% detection probability for other error models or test cases. However, the results suggest that these mechanisms may be very effective in detecting errors.

The assertions monitoring signals representing continuous values in the environment have a lower detection probability. This can be explained by the fact that these signals have more liberal constraints than the counter signals mentioned above. The liberal constraints let those errors pass which in the value domain constitute a small change in the signal, i.e. the errors most likely to remain undetected are those affecting the least significant bits of the signal. In fact, for continuous signals, errors in the least significant bits may be indistinguishable from noise in the sampling process.

The detection probability for EA7 in the signal *OutValue* was roughly 11%, whereas for all mechanisms it was 4%. This is mainly due to the fact that the behaviour of the target system is not entirely deterministic.

The results of the experiment shows that by using a number of error detection mechanisms covering different parts of the system, a fairly high total coverage may be obtained.

5.2. Total error detection probability, P_{detect}

As shown in section 2.4, the probability of detection given that an error is present in a monitored signal is part of a larger expression for total error detection probability for the entire system: $P_{detect} = (P_{en}P_{prop} + P_{em})P_{ds}$. The value obtained for P_{ds} for the target system in our evaluation was 74%. To obtain $P_{detect} = 74\%$ would mean that all the occurring errors, directly or after propagation, are uniformly distributed over the monitored signals. This is most likely not the case since there probably are some signals that are more dependent on other parts of the system than the remaining signals. If, for example, errors in our target system with a high probability propagate to the *SetValue* signal, P_{detect} would be closer to the detection probability for that signal, which in this case is roughly 59%.

From the experiments performed with error set $\boldsymbol{E_2}$, we can see that the overall detection coverage for all errors is about 10%. For errors that lead to failures, we obtained detection coverage of 39%. The values differ a lot for the two areas in which we injected errors. Generally, errors injected into the RAM area of the application were detected with a higher probability than were those injected into the stack area. An explanation for this may be that errors in the stack area more often lead to control flow errors. The evaluated mechanisms are not aimed at detecting such errors.

For the errors injected into the RAM area that eventually caused the system to fail, the detection coverage was over 81%, whereas the total detection coverage was just under 13%. We can see that if an error were of such nature that it would cause system failure we can detect it with a fairly high probability using the presented mechanisms.

5.3. Error detection latency

We can see in the results for $\boldsymbol{E_1}$ that the assertions which monitor signals that are essentially counters in nature have the shortest average detection latency. The three mechanisms that showed a 100% detection probability were also the top three mechanisms when examining the error detection latency.

Looking at the individual mechanisms shows us that the detection latencies are rather short. Most of the mechanisms had average latencies of well below one second, only mechanism EA7 had an average exceeding one second (1.604 seconds). The average of the error detection latency for all mechanisms was 511 milliseconds.

The latencies for errors in $\boldsymbol{E_2}$ are longer than the latencies for errors in $\boldsymbol{E_1}$. This, however, is not very surprising since most of the errors in $\boldsymbol{E_2}$ were not located in the monitored signals and therefore had to propagate to the monitored signals before the mechanisms could have a chance of detecting them. This propagation process increases the total time from injection to detection.

6. Summary

In this paper we investigate the properties of error detection mechanisms based on a classification scheme for signals in software. The mechanisms are generic test algorithms that are instantiated with parameters for each individual signal that is to be monitored. We have also derived an expression for the total error detection probability in a system. Two experiments were performed using error injection experiments. In the first experiment bit-flips were exercised in all bit positions of the monitored signals and in the second experiment we injected bit-flip errors in random bit positions in random memory and stack locations. The first experiment investigated the probability of detecting errors given that the errors are located in the monitored signals, as well as detection latencies. The second experiment investigated the total system detection coverage and detection latencies obtained with the mechanisms.

The detection probability was defined to be the probability of an error being detected at least once during the observation period. The detection latency was defined to be the latency between the first injected error and the first reported detection.

In the first experiment, we achieved an overall detection probability for errors in the monitored signals of 74%, and if we only take into account those errors that lead to failure we had a detection probability of over 99%. The average error detection when all mechanisms were activated simultaneously was 511 milliseconds.

The second experiment showed that for errors in the memory areas of the application we detected over 81% of all errors that caused system failure. Errors in the stack that caused system failure were detected with a probability of 13%. The low detection probability for stack errors is likely due to the fact that errors in the stack often cause control-flow errors, and the evaluated mechanisms are not aimed at detecting such errors. The detection latencies were longer than those obtained in the first experiment. This, however, is not surprising since most injected errors must propagate to the monitored signals in order to be detected. This propagation process increases the detection latency.

The presented mechanisms are good candidates for software-implemented error detection in low-cost embedded systems. They are intuitive and easy to implement and have the potential of providing high detection coverage for data errors in software signals.

Acknowledgement

We would like to thank Dr. Jörgen Christmansson and Dr. Marcus Rimén for their comments on earlier versions of this paper. We are also grateful for the comments of the anonymous reviewers, which helped to increase the quality of this paper.

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] 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

[3] 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

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

[5] Hecht H., "Fault-Tolerant Software for Real-Time Applications", *ACM Computing Surveys*, Vol.8, No. 4, pp. 391-407, December 1976

[6] Saib S.H., "Executable Assertions – An Aid To Reliable Software", *Conf. rec. 11th Asilomar Conference on Circuits Systems and Computers*, pp. 277-281, 1978

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

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

[9] Scott R.K., Gault J.W., McAllister D.F., "The Consensus Recovery Block", *Proceedings of the Total System Reliability Symposium*, pp. 74-85, 1983

[10] Kim K.H., Welch H.O., "The Distributed Execution of Recovery Blocks: An Approach to Uniform Treatment of Hardware and Software Faults in Real-Time Applications", *IEEE Transactions on Computers*, Vol. C-38, No. 5,pp. 626-636, 1989

[11] Laprie J.C., et al., "Hardware- and Software-Fault-Tolerance: Definition and Analysis of Architectural Solutions", *Proceedings of the 17th International Symposium on Fault-Tolerant Computing*, pp. 116-121, 1987

[12] Ammann P.E., Knight J.C., "Data Diversity: An Approach To Software Fault Tolerance", *IEEE Transactions on Computers*, Vol. C-37, No. 4, pp. 418-425, 1988

[13] 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

[14] 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

[15] 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

[16] 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

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

[18] Powell D., Martins E., Arlat J., Crouzet Y., "Estimators for Fault Tolerance Coverage Evaluation", *IEEE Transactions on Computers*, Vol. 44, No. 2, pp. 261-274, 1995