On the Placement of Software Mechanisms for Detection of Data Errors *

Martin Hiller, Arshad Jhumka, Neeraj Suri
Department of CE, Chalmers Univ., Göteborg, Sweden
{hiller, arshad, suri}@ce.chalmers.se
http://www.ce.chalmers.se/LDC/DEEDS

Abstract

An important aspect in the development of dependable software is to decide where to locate mechanisms for efficient error detection and recovery. We present a comparison between two methods for selecting locations for error detection mechanisms, in this case executable assertions (EA's), in black-box modular software. Our results show that by placing EA's based on error propagation analysis one may reduce the memory and execution time requirements as compared to experience- and heuristic-based placement while maintaining the obtained detection coverage. Further, we show the sensitivity of the EA-provided coverage estimation on the choice of the underlying error model. Subsequently, we extend the analysis framework such that error-model effects are also addressed and introduce measures for classifying signals according to their effect on system output when errors are present. The extended framework facilitates profiling of software systems from varied dependability perspectives and is also less susceptible to the effects of having different error models for estimating detection coverage.

1 Introduction

An integral part of developing dependable software is to incorporate error detection mechanisms (EDM's) and error recovery mechanisms (ERM's), such as containment wrappers (see, e.g., [17]), in the software structure to handle data errors (as defined in [11]). However, placement of such mechanisms has long been performed based on experience and/or heuristics (or even *ad hoc*). This has led to potential inefficient use of resources, in terms of both memory requirements and execution time, such that a higher overhead than necessary is used in order to achieve a certain level of coverage.

*Research supported in part by Volvo Research Foundation (FFP-DCN), by NUTEK (1P21-97-4745), and by Saab Endowment

In order to place EDM's and ERM's in an effective way, error propagation analysis is an important corner-piece during software development. By analyzing the propagation of errors, it is possible to find those locations in software that are likely to be subjected to errors, if there are any present in the system. In [9], we have introduced a framework for analyzing error propagation and placing EDM's/ERM's and used it together with Executable Assertions (EA's–a variant of acceptance tests, [12, 16]). In this paper, we make three specific contributions. Our first contribution is a comparison between achieved error detection coverage and resource requirements when using an experience/heuristic-based approach (EH-approach) for placement of EDM's/ERM's and when using systematic placement based on our developed error propagation analysis framework (PA-approach). The term systematic is here used to indicate that the framework (a) provides a basis of EDM/ERM placement, (b) enables quantification of the effect of good/bad EDM/ERM placement and (c) is a process for profiling and consequence assessment. Our results show that the developed error propagation analysis framework can reduce the usage of memory and execution time by reducing the number of required mechanisms while maintaining the obtained error detection coverage.

The error propagation characteristics of a software system are dependent on the considered error types, such as bit-flips, code mutations, memory leaks, etc. Errors of one type may propagate in a very different manner than errors of another type. Thus, when analyzing propagation, it is important to know what error type should be the focus of attention. In our second contribution, we illustrate how a change in error type can affect the coverage provided by the EA's placed according to our framework, showing that propagation analysis alone may not provide sufficient information for placing EDM's/ERM's, but that one must also analyse the effect of errors. An error with a low probability of propagating may still cause severe damage when it actually propagates. Our third contribution is in extending the propagation analysis framework such that the effect of varied error types can be analyzed. More specifically, we

propose measures of *impact* and *criticality* for software profiling to conduct EDM/ERM placement. These measures, taken on a signal basis, help in assessing the importance of system input or intermediate signals with respect to the system output. Overall, our intent is to provide a method for software profiling with regard to error propagation and error effect characteristics to allow effective placement of EA's for error handling.

Paper organization: We review related work in Section 2. In Section 3 we define the system model used in our proposed approach and instantiate this model with a target system described in Section 4. Two approaches for selecting locations for EA's are compared in Sections 5 and 6. In Section 7 we do a second comparison utilizing a different error model and discuss limitations in the propagation analysis framework. These limitations are addressed in an extension to the framework in Sections 8 and 9. Section 10 contains an extended analysis and profiling of the target system. Summary and conclusions are found in Section 11.

2 Related Work

Error propagation analysis for logic circuits has been in use for over 30 years. Numerous algorithms and techniques have been proposed, e.g., the D-algorithm [15], the PODEM-algorithm [6] and the FAN-algorithm [5] (which improves on the PODEM-algorithm).

Propagation analysis in software has been described for debugging use in [20]. Here the propagation analysis aimed at finding probabilities of source level locations propagating data-state errors if they were executed with erroneous initial data-states. The framework was further extended in [13, 21] for analyzing source code under test in order to determine test cases that would reveal the largest amount of defects. In [22], the framework was used for determining locations for placing assertions during software testing, i.e., aiming to place simple assertions where normal testing would have difficulties finding defects.

Finding optimal combinations of hardware EDM's based on experimental results was described in [18]. They used coverage and latency estimates for a given set of EDM's to form subsets which minimized overlapping between different EDM's, thereby giving the best cost-performance ratio.

3 Software System Model

In our studies, we consider modular software, i.e., discrete software functions interacting to deliver the requisite functionality. A module is viewed as a generalized black-box with multiple inputs and outputs. Modules communicate with each other in some specified way using varied forms of signaling, e.g., shared memory, messaging, parameter passing etc., as pertinent to the chosen communication model. We will use the term *signal* in an abstract manner, representing a software channel for data communication between modules. A system is built up from a number of such inter-linked modules. Of course, this system may be seen as a larger component or module in an even larger system.

Software systems constructed as such are found in numerous embedded systems, for example, most applications controlling physical events such as in automotive systems. Our studies mainly focus on software developed for embedded systems in consumer products (high-volume and low-production-cost systems).

In this paper we perform experiments on an example target system designed according to the described system model. The system is used for aircraft arrestment and is described in the following section.

In the subsequent sections we will utilize two different error models for our comparisons. More information on those error models will be given in conjunction with those comparisons.

4 Target: Aircraft Arrestment System

The target system is a system used for arresting aircraft on short runways The system aids incoming aircraft to reduce their velocity, eventually bringing them to a complete stop and is constructed according to specifications in [19].

4.1 Software Structure

In our study, we used actual software ported it to run on a Windows-based computer. The scheduling is slot-based and non-preemptive. Thus, from the software viewpoint, there is no difference in running on the actual hardware or running on a desktop computer.

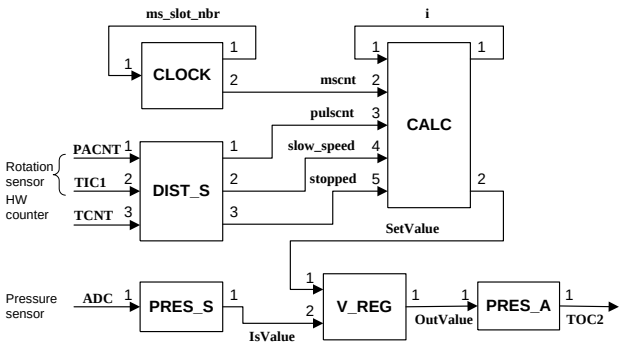


Figure 1. Software structure of target

The structure of the software is illustrated in Fig. 1. The numbers shown at the inputs and outputs are used for numbering the signals. For instance, *PACNT* is input #1 of DIST_S, and *SetValue* is output #2 of CALC.

The software is composed of six modules of varying size and input/output signal count. **CLOCK** provides a clock, *mscnt*, and a signal indicating the current execution slot for the scheduler, *ms_slot_nbr*. **DIST_S** receives *PACNT* and *TIC1* from sensors and are used to calculate the distance an aircraft has traveled on the runway, *pulscnt*. It also provides two boolean values, *slow_speed* and *stopped*, i.e., if the velocity of the aircraft is below a certain threshold or if it has stopped. **CALC** uses *mscnt*, *pulscnt*, *slow_speed* and *stopped* to calculate *SetValue*, the preferred value for the system actuators. **PRES_S** reads the value that is actually being applied by the actuators, *ADC*, and provides the signal *IsValue*. **V_REG** uses *SetValue* and *IsValue* to generate *OutValue*, the output value to the actuators. The modules attempts to compensate for the difference between *SetValue* and *IsValue*. **PRES_A** uses *OutValue* to set the actuator via the hardware register *TOC2*.

4.2 Failure Classification

The system specifications [19] set a number of physical constraints within which the system must operate. These constraints are that i) the retardation must not exceed a certain limit, ii) the brake force applied by the cable on the aircraft must not exceed a certain limit, and iii) the aircraft must be arrested before the runway ends. Any violation of these constraints is regarded as a failure. For this implementation the constraints are the following:

1. Retardation, $r < 3.5g$.
2. Retardation force, $F_{ret} < F_{max}$. The maximum allowed retardation force, F_{max}, is a function of the aircraft mass and its engaging velocity (velocity when arrestment is initiated).
3. Stopping distance, $d < 335$ meters.

Now we have presented the target system used in our comparisons. The next section will briefly describe the propagation analysis framework with associated measures used for placement of EA's and subsequent sections contain the comparison itself.

5 Ascertaining Locations for EDM's: Two Approaches

Here we illustrate how using a rigorous approach to placement of EDM's can actually decrease the requirements on memory and execution time while maintaining (and possibly increasing) the provided error detection coverage. We show two specific studies, namely: (1) the results obtained for a system where we used an experience/heuristic-based approach (which we refer to as the **EH-approach**) and (2) the results obtained using our developed error propagation analysis framework (which we refer to as the **PA-approach**) to identify locations for EDM's for the same embedded system. The EDM's used in this experiment were generic parameterized Executable Assertions (defined in [7]), EA's, which fall into the category of acceptance tests.

5.1 Placing EDM's Using Experience/Heuristics

In previous experiments on the target system [7], we had selected a number of locations in the software according to an experience/heuristic-based approach (**EH-approach**). The following process was used:

1. Identify input and output signals of the system and paths from each input signal to one or more output signals.
2. Identify internally generated signals with a direct influence on intermediate and output signals.
3. Determine which signals that are the most critical for system operation, e.g. by using FMECA.
4. Decide on locations for error detection mechanisms.

Using this process the following signals were selected as EDM-locations: *SetValue*, *IsValue*, *i*, *pulscnt*, *ms_slot_nbr*, *mscnt*, and *OutValue*. As the selection of locations in this target system was made before the described framework for propagation analysis existed, there is no bias that makes the selection consciously less good than the selection obtained after propagation analysis.

We have now selected EDM-locations based on the EH-approach. Next, we will select locations using a systematic approach based on propagation analysis, but first we give a brief overview of the analysis framework used.

5.2 Propagation Analysis: The Basic Framework

We first provide a brief description of the error propagation analysis framework (see [9] for details) used for systematic EDM/ERM-placement in this paper (**PA-approach**). The analysis framework will subsequently be extended in this paper. The framework focuses on the analysis of the propagation of data errors, i.e., erroneous values in the internal variables and signals of a system.

Figure 2. A basic black-box software module

Consider the black-box software module in Fig. 2. For each pair of input and output signals, we can define *error permeability* as the conditional probability of an error occurring on the output given that there is an error on the input. Thus, for input *i* and output *k* of a module **M** we define the *error permeability*, $P^M_{i,k}$, as follows:

$$0 \leq P^M_{i,k} = Pr\{\text{error in o/p } k | \text{error in i/p } i\} \leq 1 \quad (1)$$

This measure indicates how *permeable* an input/output pair of a software module is to errors occurring at the input. Based on this we defined the following measures:

Relative permeability (P^M) quantifies the "ability" of a module **M** to let propagating errors pass through it. The higher the permeability, the higher the chance of an error at the inputs getting through to the outputs. This is normalized with the number of input/output pairs and thus has a value between 0 and 1. The *non-weighted relative permeability* ($\hat{P}^M$) is the same measure without the normalization.
Error exposure (X^M) quantifies the level of exposure to propagating errors for module **M**. The higher the exposure, the more likely will the module be exposed to propagating errors (if there are errors in the system). Again, this is normalized to be between 0 and 1. The *non-weighted error exposure* ($\hat{X}^M$) is the same measure without the normalization.
Signal error exposure (X_s^S) is the equivalent of the *error exposure* but taken for a signal, **S**. Thus, with this measure we can identify which particular signals are more likely to be exposed to errors than others.

Note that these measures do not necessarily reflect probabilities. Rather, they are abstract measures that can be used to obtain a relative ordering across modules and signals, i.e., software profiling with regard to error propagation.

In [9], we further provided algorithms for generating the following structures to visualize of propagation paths:

Backtrack trees (BT's) illustrate the propagation paths that errors can take to get to a certain output signals. A BT has a system output signal as its root node and the various paths to system input signals, represented by the leaf nodes, as branches.
Trace trees (TT's) go in the opposite direction and depict the propagation paths from input signals to output signals. The root node represents a system input signal and the various branches are the propagation paths to any output signal, represented by the leaf nodes.

Identifying candidate locations for EDM's and ERM's is a process where trade-offs have to be weighed against each other. Given the set of measures and visualizations we have defined so far, we can now set up the following guide-lines for interpretation of obtained results:

R1: The higher the error exposure values of a module, the higher the probability that it will be subjected to errors propagating through the system if errors are indeed present. Thus, it may be more cost effective to place EDM's in those modules than in those with lower error exposure. An analogous way of reasoning is valid also for the signal error exposure.
R2: The higher the error permeability values of a module the lower its ability to contain errors. Thus, the probability of subsequent modules being subjected to propagating errors is increased. Therefore, it may be more cost effective to place ERM's in those modules than in those with lower error permeability.

Deciding on locations according to these guidelines often requires making trade-offs. For instance, one might decide to equip a module with high permeability with EDM's and ERM's even though its exposure is relatively low.

Now that we have briefly described the propagation analysis framework, we will use it in order to systematically select EDM-locations. In the subsequent sections, the two sets of selected locations and corresponding mechanisms will be compared with regard to error detection coverage and resource requirements (specifically memory and execution time requirements).

5.3 Placing EDM's Using Propagation Analysis

In the previous section we described the basic propagation analysis framework we use for systematic selection of locations for EDM's/ERM's. In this section, we will utilize that framework on our target system to select locations for Executable Assertions (EA's). Analyzing the propagation of errors using the developed framework (the PA-approach) will generate a number of estimates of permeability measures–one for each input/output pair in the target system (see Eq. 1), resulting in 25 specific estimate values.

To produce estimates of the *error permeabilities* of the modules of the target system we used fault injection (FI) as described in [9]. The FI-technique is widely used as a means for experimental estimation of f.i. coverage values (see, e.g., [1, 2, 3, 4, 10]). The permeability estimates were produced in the following way: we produced a Golden Run (GR) for each test case. Then, we injected errors in the input signals of the modules and monitored the produced output signals. For each injection run (IR) only one error was injected at one time, i.e., no multiple errors were injected. For details on the experiment setup, we refer the reader to [9].

The raw data obtained in the IR's was used in a Golden Run Comparison where the trace of each signal (input and output) was compared to its corresponding GR trace. The comparison stopped as soon as the first difference between the GR trace and the IR trace was encountered.

We only took into account the direct errors on the outputs. We did not count errors originating from errors that propagated via one of the other outputs and then came back to the original input producing an error in the first output.

Using this method we obtained the results presented in Table 1. These values are the permeability values estimated for each input/output signal pair of each module and form the basis for subsequent results.

The signal error exposure (X_s^S) for each signal (shown in Table 2) gives us better granularity for deciding which signals we should equip with EA's. The EA's we have chosen for our system are aimed at individual signals so we concentrate on the signal error exposure values and the individual permeability values when selecting locations. The selected locations are also summarized in Table 2, where also a short motivation is given as to why that particular location was selected (for details, see [9]).

Now we have selected two sets of locations for our EA's. The EH-set, i.e., the set of locations selected using the EH-approach, contains *SetValue, IsValue, i, pulscnt, ms_slot_nbr, mscnt*, and *OutValue*. The PA-set, i.e. the set of locations selected using the PA-approach, contains *SetValue, i, pulscnt*, and *OutValue*. Next, we compare, for both sets of locations, the resource requirements and the coverage obtained when the system is subjected to errors at the system inputs (e.g., by noisy and/or faulty sensors).

Input → Output	Name	Value
ms_slot_nbr → ms_slot_nbr	$P^{CLOCK}_{1,1}$	1.000
ms_slot_nbr → mscnt	$P^{CLOCK}_{1,2}$	0.000
PACNT → pulscnt	$P^{DIST\_S}_{1,1}$	0.957
TIC1 → pulscnt	$P^{DIST\_S}_{2,1}$	0.000
TCNT → pulscnt	$P^{DIST\_S}_{3,1}$	0.000
PACNT → slow_speed	$P^{DIST\_S}_{1,2}$	0.010
TIC1 → slow_speed	$P^{DIST\_S}_{2,2}$	0.000
TCNT → slow_speed	$P^{DIST\_S}_{3,2}$	0.000
PACNT → stopped	$P^{DIST\_S}_{1,3}$	0.000
TIC1 → stopped	$P^{DIST\_S}_{2,3}$	0.000
TCNT → stopped	$P^{DIST\_S}_{3,3}$	0.000
ADC → IsValue	$P^{PRES\_S}_{1,1}$	0.000
i → i	$P^{CALC}_{1,1}$	1.000
mscnt → i	$P^{CALC}_{2,1}$	0.000
pulscnt → i	$P^{CALC}_{3,1}$	0.494
slow_speed → i	$P^{CALC}_{4,1}$	0.000
stopped → i	$P^{CALC}_{5,1}$	0.013
i → SetValue	$P^{CALC}_{1,2}$	0.056
mscnt → SetValue	$P^{CALC}_{2,2}$	0.530
pulscnt → SetValue	$P^{CALC}_{3,2}$	0.000
slow_speed → SetValue	$P^{CALC}_{4,2}$	0.892
stopped → SetValue	$P^{CALC}_{5,2}$	0.000
SetValue → OutValue	$P^{V\_REG}_{1,1}$	0.885
IsValue → OutValue	$P^{V\_REG}_{2,1}$	0.896
OutValue → TOC2	$P^{PRES\_A}_{1,1}$	0.875

Table 1. Estimated error permeability values of the input/output pairs

Signal	X_s^S	Select	Motivation
OutValue	1.781	yes	High error exposure
i	1.507	yes	High error exposure
SetValue	1.478	yes	High error exposure
ms_slot_nbr	1.000	no	Zero error permeability to mscnt
pulscnt	0.957	yes	High error exposure
TOC2	0.875	no	Errors here most likely come from OutValue
slow_speed	0.010	no	Low error exposure, selected EA's not geared at boolean values
IsValue	0.000	no	Zero error exposure
mscnt	0.000	no	Zero error exposure
stopped	0.000	no	Zero error exposure

Table 2. Estimated signal error exposures and PA-based selection of EA locations

Signal	EA	EH-set	PA-set	ROM (bytes)	RAM (bytes)
SetValue	EA1	√	√	50	14
IsValue	EA2	√	-	50	14
i	EA3	√	√	25	13
pulscnt	EA4	√	√	25	13
ms_slot_nbr	EA5	√	-	37	13
mscnt	EA6	√	-	25	13
OutValue	EA7	√	√	50	14
Total ROM/RAM (bytes)		262/94	150/54		

Table 3. EA-setup and sum of RAM/ROM requirements

6 Comparing the Two Placement Techniques

This section will compare the two sets of selected locations with regard to the resources required and also with regard to the coverage obtained when the system is subjected to errors at the system input signals.

6.1 Memory and Execution Time Requirements

For comparison of resource requirements using the EH-approach and the PA-approach, Table 3 presents a summary of the two sets of locations/mechanisms and their respective requirements on memory resources (ROM contains constant parameters defining allowed behavior, and RAM contains run-time data). As expected, the requirements for the PA-set, {EA1, EA3, EA4, EA7}, is less than the requirements for the EH-set, {EA1, EA2, EA3, EA4, EA5, EA6, EA7}, as the former is a subset of the latter (as seen in Table 3). Specifically, there is a 40 percent reduction in memory requirements when for the PA-set over the EH-set.

The overhead in terms of execution time is also reduced. The tool used for obtaining these results does not provide a means for measuring execution time, thus we were not able to quantitatively assess the reduction. However, the EA's are all functions which are executed sequentially, i.e. the software is not executed in a truly parallel manner as only one processor is used. Also, they are invoked with roughly the same period and require roughly the same execution time for each invocation. Thus, the reduction in execution time overhead is likely to be in the order of the reduction in number of EA's, i.e., about 40 percent.

6.2 Error Detection Coverage

When comparing the error detection coverage for the two sets of EA's we assumed that the system would only be subjected to errors introduced via the system inputs, e.g., from noisy or faulty sensors. Thus we assumed errors to get into the system via these four signals: 1) *ADC*, 2) *PACNT*, 3) *TIC1*, and 4) *TCNT*.

For the error injection experiments, we used the same setup, i.e. the same error model and the same test cases, as we used for the propagation analysis.

As we already have seen, errors (from this error model) injected into *ADC* did not propagate further into the system, so we can safely say that the coverage of detection for those errors will be zero (as there is nothing to detect) for both sets of EA's and locations. Thus, we can concentrate our experiments on the remaining three signals.

In Table 4 we summarize the results from the injection experiments. The results are shown for each input signal that was targeted during the experiments. The n_{err} column shows how many errors that were active after injection (e.g., we injected a total of 2,000 errors in *PACNT*, and of

Signal	n_{err}	EA1	EA2	EA3	EA4	EA5	EA6	EA7	Total
Member of EH-set		√	√	√	√	√	√	√	
Member of PA-set		√	-	√	√	-	-	√	
PACNT	1856	0.218	0.105	-	0.975	-	-	0.005	0.975
TIC1	3712	-	-	-	-	-	-	-	-
TCNT	3712	-	-	-	-	-	-	-	-
All	9280	0.062	0.040	-	0.195	-	-	< 0.001	0.195

Table 4. Obtained detection coverage for errors injected in system input - EH-based and PA-based EA-placements

those 1,856 were active, i.e., injected before the arrestment of an aircraft was not completed). The various *EAx* columns show the obtained coverage for each individual EA (a dash indicates zero coverage), calculated as n_{det}/n_{err}. The *Total* column is the combined coverage considering all EA's. Each row contains the data for errors injected into one signal except for *All* which shows the coverage obtained for the various EA's considering all signals. The rows containing tick-marks indicate which EA's were part of the EH-set and of the PA-set (a tick-mark indicates membership).

In Table 4 we can see that only those errors that were injected into *PACNT* were detected. This is on par with the results obtained in the propagation analysis which showed that errors injected into those signals with a very low probability propagated into any of the signals that were selected to be guarded with an EA. Those errors that propagate are likely to be hard to detect by the selected mechanisms. However, 97.5 percent of the errors injected into *PACNT* were detected. All errors detected by EA1, EA2 or EA7 were also detected by EA4.

It may seem odd that EA2, which guards *IsValue*, has a non-zero coverage for errors in *PACNT*, while no errors injected into *ADC* could propagate into *IsValue*. This, however, is a result of errors in *PACNT* propagating all the way through the system and out beyond the system barrier where they eventually affect the environment to such a degree that *ADC* is affected in a way the PRES_S module cannot fully mask or contain, and the errors are then detected by the EA guarding *IsValue*.

Comparing the results shown in Table 4 also shows that the obtained coverage for the two sets of EA's is the same.

6.3 EH v/s PA: Comparative Summary

Now we have resource and coverage information for the various EA's which enables us to perform the comparison between the two sets of selected locations and mechanisms.

Starting with the resources from Table 3, we can see that the requirements on memory (i.e. RAM and ROM combined) for EA's in the the EH-set is almost double that of those in the PA-set.

Furthermore, from Table 4 we can observe that the coverage obtained with the EH-set of EA's (EA1 through EA7) is the same as that obtained with the PA-set set of EA's (EA1, EA3, EA4, and EA7).

From this we can conclude that the PA-approach successfully enabled us to cut the resources used for error detection while maintaining the detection coverage. The next step in our comparison will investigate the effect of varying error model on the obtained error detection coverage.

7 The Effect of Varying Error Model On Error Detection Coverage

At this point we have shown that when assuming that errors are introduced into the system via the system inputs the framework can enable developers to reduce the amount of resources required in order to obtain a certain level of dependability. In this section we will investigate the ramifications of changing the error model. The error model used in the previous sections for ascertaining coverage values may be considered a "nice" error model in that it only has a direct effect on the input signals of the system and only disrupts these once during the entire duration of the arrestment.

Here we change the error model to a more severe one. We still use single bit flips to generate data errors, but now the target will not only be system input signals but also intermediate signals and module state (a total of 150 locations in RAM and 50 locations in the stack) of the system. The errors are injected not only at one point in time but periodically with a period of 20 ms. The same 25 test cases were used giving us a total of 200·25 = 5000 arrestment with injections. An error is said to be detected if it is detected at least once during the arrestment.

The results are summarized in Fig. 3. The *RAM*-bars are the coverage values for errors injected into the RAM areas of the modules, and the *Stack*-bars the coverage values for errors injected into the stack area. The *Total*-bars are the coverage for all errors. The measure c_{tot} is the total coverage of the EA-set, c_{fail} is the coverage considering only errors that led to system failure (according to the classification of Section 4), and c_{nofail} is for errors that did not lead to system failure.

The first observation we make when comparing the re-

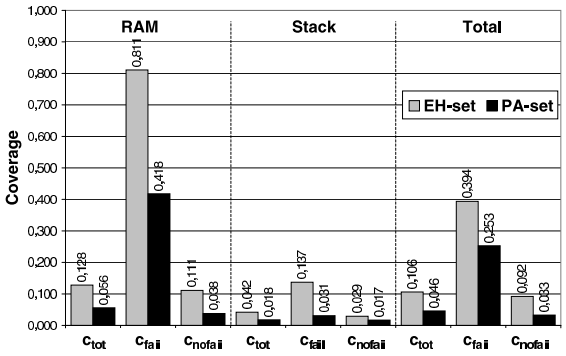


Figure 3. Comparison of coverage values

sults for the two approaches is that the coverages for the PA-set of EA's are lower than the coverage for the EH-set. For errors injected into RAM the coverage is just over half that obtained using the full set of EA's and for errors in stack the decrease is even greater. This indicates a sensitivity in the propagation analysis framework when it comes to dealing with error models where errors are introduced not only via the inputs of a system, but rather via internal variables and structures.

Thus, it may not be sufficient to only take into account error propagation, but that one also has to consider the *what if*'s that may have a very low probability of occurring but still have a profound effect on system operation should they actually occur. To address these limitations in our existing framework, we extend it with additional measures adding error effect analysis to the provided error propagation analysis. This has the property that the resulting framework is made more resilient to differences in error model. The extensions are described in the following sections.

8 Effect Analysis: Software Profiling With Regard to Error Impact and Criticality

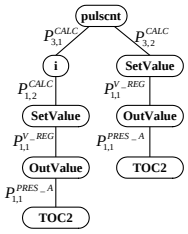
From the results shown in Section 7, we can see that propagation analysis alone may be insufficient when selecting locations for EDM's and ERM's. Errors that may have a low probability of propagating may still cause severe damage should propagation occur. Taking this into account we now define measures which let us analyse to what extent errors in a signal (system input signal or intermediate signal) affect the system output, i.e., what is the impact of errors on the system output signals.

As errors in a source signal can propagate along many different paths to the (destination) system output signal we must consider this in our definition of impact. In order to calculate the impact of errors in a signal S_s on a system output signal S_o we must first generate an *impact tree*, which is a generalization of trace trees (see Section 5). This tree has the signal of interest as the root, in this case S_s. Next, we generate all the propagation paths from the root to the leaves containing system output signal S_o (there may be leaves generated by other system output signals). Each path i has a weight w_i associated with it which is the product of all permeability values along that path. We define $S_s \rightsquigarrow S_o$, the *impact* of (errors in) S_s on S_o, as

$$0 \leq S_s \rightsquigarrow S_o = 1 - \prod_i (1 - w_i) \leq 1 \qquad (2)$$

where w_i is the weight of path i from S_s to S_o. If one could assume independence all over, the impact measure would be the conditional probability of an error in S_s propagating all the way to S_o. However, as independence can rarely be assumed we will treat this as a relative measure by which different signals can be ranked. The general interpretation of this measure is that the higher the impact, the higher the risk of an error in the source signal generating an error in the output of the system. Thus, when placing EDM's and ERM's one may consider placing such mechanisms at signals which have a high impact even though they may have a low error exposure (meaning that errors in this signal are relatively rare but costly, should they occur).

To illustrate this further, consider our target system shown in Fig. 1. Suppose we would like to calculate the impact of errors in signal *pulscnt* on system output *TOC2*. First, we will generate an impact tree as shown in in Fig. 4.



Paths generated from the impact tree:

$w_1 = P^{CALC}_{3,1} P^{CALC}_{1,2} P^{V\_REG}_{1,1} P^{PRES\_A}_{1,1} = 0.021$

$w_2 = P^{CALC}_{3,2} P^{V\_REG}_{1,1} P^{PRES\_A}_{1,1} = 0.000$

Figure 4. Impact tree for signal *pulscnt* and generated propagation paths.

In order to calculate the impact of errors in *pulscnt* on system output *TOC2* we generate all the propagation paths from the root to the leaves. In this case, where we have only one system output, all leaves are considered. This gives us two paths as shown in Fig. 4.

Using the weights of the paths we can now calculate *pulscnt* $\rightsquigarrow$ *TOC2*, i.e., the impact of (errors in) *pulscnt* on *TOC2*, as

$$pulscnt \rightsquigarrow TOC2 = 1 - \prod_{i=1}^{2} (1 - w_i) = 0.021$$

where w_i are the weights shown in Fig 4.

The concept of impact as described above considers the impact on system output generated by errors in system input signals and intermediate signals. However, when a system has *multiple* outputs, these outputs are not necessarily all equally important for the operation of the system, i.e., some outputs may be more critical than others. For cost-efficiency, one may wish to concentrate resources for dependability on the most critical system outputs and therefore needs to know which signals in the system that are "best" (in a loose sense) to equip with EDM's/ERM's.

For each system output signal $S_{o,i}$ (the ith output signal) we assign a criticality value $C_{o,i}$ which is a value between 0 and 1 where 0 denotes *not at all critical* and 1 denotes *highest possible criticality*. These criticality values are assigned by the system designer for example from the specifications of the system or from results from experimental vulnerability analyses.

The criticality of system input signals and intermediate signals is calculated using the assigned criticality values of the system output signals and the various impact values calculated for the various signals. Each signal S_s has a certain impact, $S_s \rightsquigarrow S_{o,i}$, on system output $S_{o,i}$, as calculated according to Eq. 2. Based on the impact, the *criticality* of S_s as experienced by system output $S_{o,i}$ is calculated as

$$0 \leq C_{s,i} = C_{o,i} \cdot (S_s \rightsquigarrow S_{o,i}) \leq 1 \qquad (3)$$

Once we have the criticality of S_s with regard to each system output signal $S_{o,i}$ we can calculate a total criticality. We define the *criticality* C_s of signal S_s as

$$\begin{aligned} 0 \leq C_s &= 1 - \prod_i (1 - C_{s,i}) \\ &= 1 - \prod_i (1 - C_{o,i} \cdot (S_s \rightsquigarrow S_{o,i})) \leq 1 \\ &\leq 1 \end{aligned} \qquad (4)$$

The criticality measure indicates for each signal how "expensive" errors are with regard to the total system operation, i.e., the higher the criticality value, the higher the likelihood of the system not being able to deliver its intended service, should an error occur in the signal. The notion of criticality as defined here also takes into account the "cost" associated with errors in system outputs as defined by the system designer. Thus, while the impact measures are independent of the project policies regarding dependability, the criticality values may change when project policies change.

Note that if the system only has one output signal then the criticality will only function as a constant scaling factor, i.e., the relative order among the signals of the system will not change. Thus, calculating criticality values is only required when there are multiple output signals in a system. If there are multiple outputs, a given signal may have identical impact for different outputs, but the criticalities may differ.

In the following section we discuss how the various measures obtained in the error effect analysis together with the values obtained in the error propagation analysis can be used to identify candidate locations for EDM's and ERM's.

9 Candidate Locations For EDM's/ERM's

In this section we discuss how candidate locations for EDM's and ERM's may be identified based on the results from the propagation analysis and the effect analysis. We have, in Section 5, given two rules of thumb for the propagation analysis. Here we add one for effect analysis:

R3: The higher the criticality (or impact if the system only has one output signal) of a signal, the higher the probability of an error in that signal causing damage from a system point-of-view. Thus, it may be more cost effective to equip those signals with EDM's and ERM's which have the highest criticality (impact).

When selecting locations, the rules of thumb (R1, R2 and R3) may have to be weighed against each other. Consider the case where a signal has a *low exposure* but a *high criticality*. The low exposure means that there is a low probability of errors propagating to that signal. However, the high criticality means that, should an error find its way into that signal, there is a high probability of that error causing damage which propagates beyond the system barrier into the environment. Thus, one may select signals with low exposure and high criticality as candidate locations for EDM's and ERM's.

As a process, a possible approach to placement of EDM's and ERM's may be to set up specific conditions which the software must conform to. For example, one may wish to set a minimum level of error containment for all modules, which can be accomplished by setting a maximum level on error permeability values. Thus, a module exceeding this limit indicates that more resources have to be allocated to that module to increase its error containment capabilities. The same argument can be used for error exposure. If a module or signal is highly exposed, this indicates that more resources are required either to protect the exposed module or signal, or to increase the error containment capabilities of the module or signal responsible for the high degree of exposure.

From a criticality (impact) point-of-view, a project may also set up certain limits. For example, one may wish to set a maximum level of impact for the various signals. Signals exceeding this limit indicate that the error containment from that signal out to the system output signals is not high enough. As the criticality values of signals are based on the criticality values assigned to system output signals, these can only be indirectly adjusted via the impact values.

The results from the analysis may also aid in the design of EDM's. For example, a situation with low error exposure and high criticality (impact) indicates that any EDM in

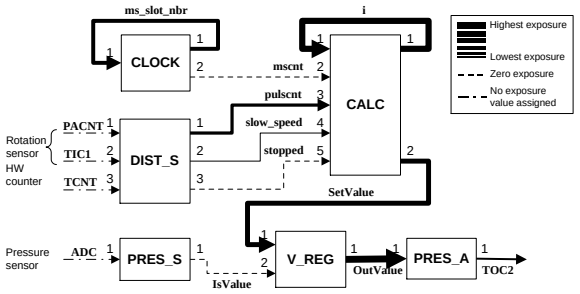


Figure 5. Exposure profile of target system

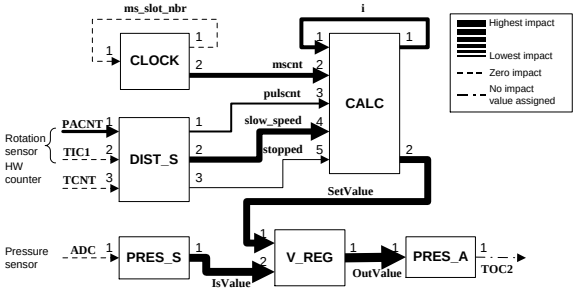


Figure 6. Impact profile of target system

that location would have to be highly specialized as errors are infrequent and likely to be hard to detect. The opposite situation, i.e., high exposure and low criticality (impact) indicates that a coarser EDM in that location may suffice.

Having extended the analysis framework, we refer back to our target system and conduct effect analysis to see if there are more locations for EDM's/ERM's to be selected.

10 Extended Analysis of Target System

Using the permeability values presented in Table 1 we can now obtain impact values for the various signals of the target system. As the target system has only one output signal, there is no need to produce any criticality values as they would be the same as the impact values, only adjusted with a constant. Table 5 contains these impact values together with the previously obtained signal exposure values.

Signal (s)	X_s^S	$s \rightsquigarrow$ *TOC2*
PACNT	-	0.027
TCNT	-	0.000
TIC1	-	0.000
ADC	-	0.000
OutValue	1.781	0.875
i	1.507	0.043
SetValue	1.478	0.774
ms_slot_nbr	1.000	0.000
pulscnt	0.957	0.021
TOC2	0.875	-
slow_speed	0.010	0.691
IsValue	0.000	0.784
mscnt	0.000	0.410
stopped	0.000	0.001

Table 5. Estimated signal error exposures and impacts on *TOC2*

In Table 5, we now have both exposure values, X_s^S, and impact values, $s \rightsquigarrow$ *TOC2*, of the various signals of the target system. Signal *TOC2* has no impact value associated with it as this is the system output signal (one could say that the impact is 1.0 in this case). The same information is depicted graphically in Figs. 5 and 6. Here we can clearly see the difference between the two profiles of the system. The thickness of a line now depicts the value of the respective measure–the thicker the line, the higher the value. A dashed line indicates a zero value and a dashed-dotted line indicates that no value is assigned to that signal (either because the signal is a system input or output signal).

Previously we had ascertained that signals *SetValue*, *i*, *pulscnt* and *OutValue* were to be guarded by EA's because of their high exposure to propagating errors. If we now take into account the impact of the signals on system output, we see that signals *IsValue*, *mscnt* and *slow_speed* may be considered for being guarded by EA's as well. The mechanisms we have chosen are implemented such that it is difficult to detect errors in a boolean value, thus setting an EA on the signal *slow_speed* is not efficient in this case. Therefore, when taking into account also the impact values of the signals we can decide to place EA's on *IsValue* and *mscnt* as well. Also, as the permeability-value of *ms_slot_nbr* is 1, and the assumed error model now introduces errors in the entire memory space of the system (as opposed to only system input signals earlier) we also select that signal.

The results illustrate the important distinction between permeability/exposure and impact/criticality, where the former is used for profiling software with regard to its error propagation characteristics and the latter to profile software with regard to the effect errors would have if they were present in different parts of the system.

In this specific example, using the locations indicated by the extended framework will give us the same coverage values as for the EH-approach (see Fig. 3), as we now have the same set of EA's selected by the EH-approach and the extended framework. This shows that with the extension presented here the analysis framework is more robust with regard to the error model, i.e., we have reduced the effect of the error model on the analysis results obtained with the framework.

11 Summary and Conclusions

The focus of this paper was on presenting methods for selecting locations in embedded modular software where er-

ror detection and recovery mechanisms (EDM's and ERM's) should be considered in order to increase dependability with regard to data errors. We make the following contributions:

C1: We compared two approaches for selecting locations in software for EDM's/ERM's: i) an approach based on experience/heuristics (EH-approach), and ii) an approach based on propagation analysis (PA-approach, [9]). The comparison used the software of an aircraft arrestment system and showed that the PA-approach could decrease, compared to the EH-approach, the resource requirements (memory and execution time) for Executable Assertions (EA's), while maintaining the obtained error detection coverage.

C2: The error model in **C1** introduced errors into the system only via its main input signals. A second comparison, using an error model where errors were introduced also in the internal variables and structures, illustrated that the locations selected using the PA-approach to be guarded by EDM's in **C1** did not obtain the same coverage as the locations selected by the EH-approach. The results show that in addition to analyzing the propagation characteristics of errors in software, one must also take into account the effect of errors–an error may have a very low probability of propagating into a given signal, but should this signal contain an error this may have severe consequences on the output of the system. This points out a limitation in the PA-approach which does not take error effect into account.

C3: In order to remove the limitation identified in **C2**, we extended the framework with means to analyse the effect of errors by adding the measures *impact* and *criticality*. The *impact* is a measure of the probability of an error in a signal to disrupt the system output. When a system has multiple outputs with varying "importance" (e.g., a diagnostic output may not be as "important" as an actuator control output), the *criticality* measure allows the developer to rank outputs and consequently scale the impact values of signals. Thus, two signals with the same impact may have different criticalities depending on which outputs they affect the most. Using the extended analysis framework we could select locations for EDM's such that high error detection coverage was obtained also for the more disruptive error model used in **C2**.

From this we can conclude that the extended error propagation and effect analysis framework provides software developers with a generalized process to profile modular software systems such that dependability resources can be allocated according to system requirements and available resources. We comment that our software profiling approach is developed for generic modular black-box software, ensuring its scalability with regard to software size.

Future work includes applying the analysis framework on alteranet target systems in order to validate the generalized applicability of the obtained results. We will also look at using the framework in conjunction with methods for assessing overall consistency and coverage of EA's.

12 Acknowledgments

We would specifically like to thank Professor Inhwan Lee for his help and comments over the paper revision, and also the entire DEEDS group at Chalmers for their multi-faceted support.

References

[1] Arlat, J., et al., "Fault Injection for Dependability Validation: A Methodology and Some Applications", *IEEE Trans. on SE*, Vol. 16, No. 2, pp. 166-182, 1990.

[2] Chillarege R., Bowen N. S., "Understanding Large System Failures - A Fault Injection Experiment", *Proc. FTCS-19*, pp. 356-363, 1989.

[3] Cukier M., et al., "Coverage Estimation Methods for Stratified Fault-Injection", *IEEE Trans. on Comp.*, pp. 707-723, 1999.

[4] Fabre J.-C., et al., "Assessment of Microkernels by Fault Injection", *Proc. DCCA-7*, pp. 25-44, 1999.

[5] Fujiwara H., Shimono T. "On the Acceleration of Test Generation Algorithms", *Proc. FTCS-13*, pp. 98-105, 1983.

[6] Goel P., "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits", *IEEE Trans. on Comp.*, Vol. 30, No. 3, pp. 215-222, 1981.

[7] Hiller M., "Executable Assertions for Detecting Data Errors in Embedded Control Systems", *Proc. DSN 2000*, pp. 24-33, 2000.

[8] Hiller M., "A Tool for Examining the Behavior of Faults and Errors in Software", *TR 00-19*, Dept. of CE, Chalmers Univ., (available at http://www.ce.chalmers.se/LDC/DEEDS/), 2000.

[9] Hiller M., Jhumka A., Suri N., "An Approach for Analysing the Propagation of Data Errors in Software", *Proc. DSN 2001*, pp. 161-170, 2001.

[10] Iyer R. K., Tang D., "Experimental Analysis of Computer System Dependability", Chapter 5 in *Fault-Tolerant Computer System Design* (ed. D.K. Pradhan), Prentice Hall, 1996.

[11] Laprie J.-C., "Dependable Computing: Concepts, Limits, Challenges", *Proc. FTCS-25*, pp. 42-54, 1995.

[12] Mahmood A., et al., "Executable Assertions and Flight Software", *Proc. DASC-6*, pp. 346-351, 1984.

[13] Morell L., Murrill B., Rand R., "Perturbation Analysis of Computer Programs", *Proc. COMPASS'97*, pp. 77-87, 1997.

[14] Powell D., et al., "Estimators for Fault Tolerance Coverage Evaluation", *IEEE Trans. on Comp.*, Vol. 44, No. 2, pp. 261-274, 1995.

[15] Roth J.P., *Computer Logic, Testing and Verification*, Computer Press, 1980.

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

[17] Salles F., et al., "MetaKernels and Fault Containment Wrappers", *Proc. FTCS-29*, pp. 22-29, 1999.

[18] Steininger A., Scherrer C., "On Finding an Optimal Combination of Error Detection Mechanisms Based on Results of Fault Injection Experiments", *Proc. FTCS-27*, pp. 238-247, 1997.

[19] US Air Force - 99, "MIL-SPEC: Aircraft Arresting System BAK-12A/E32A; Portable, Rotary Friction", MIL-A-38202C, Notice 1, US Dept. of Defense, Sept. 2, 1986.

[20] Voas J., Morell L. J., "Propagation and Infection Analysis (PIA) Applied to Debugging", *Proc. of Southeastcon'90*, pp. 379-383, 1990.

[21] Voas J., "PIE: A Dynamic Failure-Based Technique", *IEEE Trans. on SE*, Vol. 18, No. 8, pp. 717-727, 1992.

[22] Voas J., et al., "Error Propagation Analysis Studies in a Nuclear Research Code", *Aerospace Conf.*, Vol. 4, pp. 115-121, 1998.