Software Profiling for Designing Dependable Software *

Martin Hiller
Department of Electronics & Software
Volvo Technology Corporation
Göteborg, Sweden
martin.hiller@volvo.com

Arshad Jhumka Neeraj Suri
Department of Computer Science
Technische Universität Darmstadt
Darmstadt, Germany
{arshad,suri}@informatik.tu-darmstadt.de

Abstract

This paper describes a method for profiling modular software by analyzing the propagation and effect of data errors. A framework of different metrics, based on the concept of error permeability *enables the profiling of vulnerabilities and hot-spots, specifically i) the modules and signals that are most likely exposed to propagating errors, and ii) the modules and signals which, when subjected to errors, tend to cause more damage than others. Based on the profiles obtained by the metrics framework we discuss how to identify where dependability structures and mechanisms are likely to be the most effective. We also describe a fault-injection-based method for estimating the metrics, and profile the software of a real embedded system to show the type of results obtainable by the framework.*

1 Introduction

The increasing use of software in computerized systems to lower costs for certain functionalities has produced a large increase in the need for software implemented mechanisms for dependability. An integral part of developing low-cost robust systems is to equip software with structures and mechanisms providing dependability (e.g., masking fault tolerance, fail safety or fail silence). The necessary prerequisites for developing and providing the relevant dependability in software are:

1. The type of errors that the system is supposed to handle; their nature, frequency, duration, etc. If this is not known, it is very hard to know how to obtain any dependability. This would make both the development as well as the assessment/analysis of the system difficult (if not impossible).
2. The available structures and mechanisms for dependability. When developing dependable software it is of course important to know the characteristics and properties of the mechanisms at ones disposal, including their strengths and weaknesses. The overall architecture of the software may be affected by these properties.
3. The vulnerabilities and hot-spots of the software. In order to be able to incorporate dependability structures and mechanisms where they are likely to be the most effective, it is important to know where errors tend to propagate and where errors tend to do the most damage.

This paper has a focus on the last issue above and presents a framework for profiling modular software with regard to error propagation and error effect. The framework is called *EPIC* after the four groups of measures it introduces (*Exposure*, *Permeability*, *Impact*, and *Criticality*).

Propagation analysis may be used to find the modules and signals which are most exposed to errors in a system, and to ascertain how different modules affect each other in the presence of errors. In addition to knowing error propagation characteristics it is also important to know where errors are likely to do the most damage. Note that those errors which are most likely to propagate are not always those that are most likely to cause great damage. Thus it is important to do an analysis of both notions to identify the most vulnerable parts of a system.

The focus of *EPIC* is on handling data errors and we consider modular software resident on either single or distributed hardware nodes. In our approach, we adopt a black-box view of modular software and introduce the measure *error permeability* as well as a set of related measures. Subsequently, we define a methodology for using these measures to obtain software profiles providing information on error propagation and error effect, and also aid in identifying hot-spots and vulnerabilities in the software. As such, parts of the calculations and techniques used in the framework re-

*This work was mainly performed at Chalmers University of Technology (Göteborg, Sweden), supported in part by Volvo Research Foundation (FFP-DCN), by NUTEK (1P21-97-4745), by Saab Endowment, and by NSF Career CCR 9896321

semble those used for reliability diagrams and fault trees.

Paper organisation: Section 2 reviews related work and Section 3 describes the assumed system model and introduce an example system. The EPIC framework is described in Section 4. In Section 5, a method for estimating numerical values of the metrics is discussed, and we also produce estimates of the various metrics for our example system. Section 6 contains a summary and conclusions.

2 Related Work

Error propagation analysis for logic circuits has been in use for many decades. Numerous algorithms and techniques have been proposed, e.g., the D-algorithm [9], the PODEM-algorithm [3] and the FAN-algorithm [2] (which improves on the PODEM-algorithm).

Propagation analysis in software has been described for debugging use in [11]. 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 [7, 12] for analysing source code under test in order to determine test cases that would reveal the largest amount of defects. In [13], the same 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.

3 Assumed Models, and Example System

System/Software Model, and Fault 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 generalised 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.

The fault model which EPIC is aimed at is that of data errors. That is, errors in variables and signals. However, we do not explicitly consider data errors which may result in control error errors.

An Example Embedded System In order to illustrate the proposed methodology of profiling error propagation and effect in software, we use the software of an embedded control system used for arresting aircraft on short runways and aircraft carriers, constructed according to specifications found in [10].

In our study, we used actual software of the system master and 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.

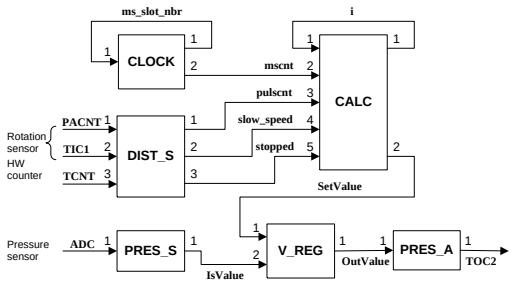

Figure 1. Software structure of target

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 millisecond-clock, *mscnt*. The signal *ms_slot_nbr* indicates the current execution slot. **DIST_S** receives *PACNT* and *TIC1* from the rotation sensor and *TCNT* from the hardware counter modules. The rotation sensor reads the number of pulses generated by a tooth wheel on the drum. The module provides a total count of the pulses, *pulscnt*, generated during the arrestment. It also provides two Boolean values, *slow_speed* and *stopped*. **CALC** uses *mscnt*, *pulscnt*, *slow_speed* and *stopped* to calculate a set point value for the pressure valves, *SetValue*, at six predefined checkpoints along the runway. The checkpoints are detected by comparing the current *pulscnt* with pre-defined values corresponding. The current checkpoint is stored in *i*. **PRES_S** reads the the pressure that is actually being applied by the pressure valves, using *ADC* from the internal A/D-converter. This value is provided in *IsValue*. **V_REG** uses *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*. **PRES_A** uses *OutValue* to set the pressure valve via the hardware register *TOC2*.

4 Software Profiling using EPIC

The EPIC framework aims at providing a means of profiling software such that weaknesses and hot-spots in modular software can be identified. To achieve this, EPIC can be used to generate two distinct profiles of a software system: i) error propagation profile and ii) error effect profile. These chart how data errors propagate through a software system and their effect on system operations, respectively. When performing a cost/benefit analysis, both profiles are used.

4.1 Error Permeability - Letting Errors Pass

The basis of the approach is *error permeability*. Upon this, we define a set of related metrics providing an insight on the error propagation and effect characteristics of a system.

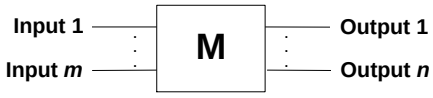

Figure 2. A basic black-box software module with m inputs and n outputs

Consider the software module in Fig. 2. For each pair of input and output signals, the *error permeability* is defined 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**, the *error permeability*, $P^M_{i,k}$, is defined as follows:

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

This measure indicates how *permeable* an input/output pair of a software module is to errors occurring on that particular input. It should be noted that if the error permeability of an input/output pair is zero, this does not necessarily mean that the incoming error did not cause any damage. The error may have caused a latent error in the internal state of the module that for some reason is not visible on the outputs. In Section 5, we describe an approach for experimentally estimating values for this measure.

Going to the module level (Fig. 2), we define the *module error permeability*, P^M, of a module **M** with m input signals and n output signals, to be:

$$0 \leq P^M = \left(\frac{1}{m} \cdot \frac{1}{n}\right) \sum_i \sum_k P^M_{i,k} \leq 1 \quad (2)$$

In order to be able to make a distinction between modules with a large fan in/out and those with a small fan in/out, we can, for a module **M** with m input signals and n output signals, define the *non-normalized module error permeability*, $\hat{P}^M$ as follows:

$$0 \leq \hat{P}^M = \sum_i \sum_k P^M_{i,k} \leq m \cdot n \quad (3)$$

The two measures defined in Eqs. 2 and 3 are both necessary for analyzing the modules of a system. For instance, consider the case where two modules, **G** and **H**, are to be compared. **G** has few inputs and outputs, and **H** has many. Then, if $P^G = P^H$, then $\hat{P}^G < \hat{P}^H$. And vice versa, if $\hat{P}^G = \hat{P}^H$, then $P^G > P^H$.

4.2 Ascertaining Propagation Paths

In order to gain knowledge about the exposure of the modules to propagating errors in the system we define the following process which considers interactions across modules.

Consider the example software system shown in Fig. 1. External input to the system is received at *PACNT*, *TIC1*, *TCNT*, and *ADC*. The output is *TOC2*.

Once we have obtained values for the error permeability for each input/output pair of each module, we can construct a *permeability graph* as illustrated in Fig. 3.

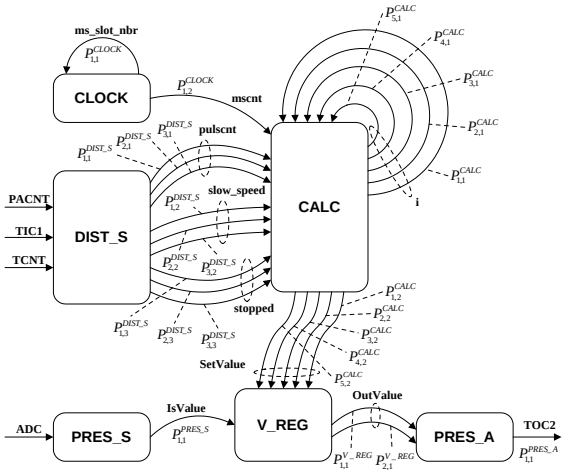

Figure 3. Permeability graph of target

In the graph (Fig. 3) we can see the various permeability values (labels on the arcs) that will have to be calculated. The numbers used in the notation refer to the numbers of the input signals and output signals respectively, as shown in Fig. 1. For instance, $P^{CALC}_{2,1}$ is the error permeability from input 2 (*mscnt*) to output 1 (*i*) of module CALC.

With the permeability graph we can:

- **A** Backtrack from system output signals to system input signals in order to find those paths which have the highest probability of error propagation (*Output Error Tracing*), or
- **B** Trace errors from system input signals to system output signals in order to find which paths these errors will most likely propagate along (*Input Error Tracing*).

Output Error Tracing is easily accomplished by constructing a set of *backtrack trees*, one for each system output, according to the procedure in Fig. 4.

This will, for each system output, give us a backtrack tree where the root corresponds to the system output, the intermediate nodes correspond to internal outputs and the leaves correspond to system inputs (or module inputs receiving feedback from its own module). Also, all vertices in the tree have a weight corresponding to an error permeability value. Once we have obtained this tree, finding the propagation paths with the highest propagation probability is simply a matter of finding which paths from the root to the leaves have the highest weight.

A1. Select a system output signal and let it be the root node of the backtrack tree.
A2. For each error permeability value associated with the signal, generate a child node that will be associated with an input signal.
A3. For each child node, if the corresponding signal is not a system input signal, backtrack to the generating module and determine the corresponding output signal. Use this signal and construct the sub-tree for the child node from A2. If the corresponding signal is a system input signal it will be a leaf in the tree. If the corresponding signal is an input signal to the same module it will be a leaf in the tree (as opposed to other leaves which are system input signals). We do not follow the recursion that is generated by the feedback.
A4. If there are more system output signals, go back to A1.

Figure 4. Generating backtrack trees.

Input error tracing is achieved similarly. Here, we construct a *trace tree* for each system input, according to the procedure in Fig. 5.

B1. Select a system input signal and let it be the root node of the trace tree.
B2. Determine the receiving module of the signal and for each output of that module, generate a child node. This way, each child node will be associated with an output signal.
B3. For each child node, if the corresponding signal is not a system output signal, trace the signal to the receiving module and determine the corresponding input signal. Use this signal and construct the sub-tree of the child node from B2. If the corresponding signal is a system output signal it will be a leaf in the tree. If the input signal is the same module that generated the output signal (i.e. we have a module feedback) then follow this feedback once and generate the sub-trees for the remaining outputs. We do not follow the recursion generated by this feedback.
B4. If there are more system input signals, go back to B1.

Figure 5. Generating trace trees.

This procedure results in a set of trace trees - one for each system input. In a trace tree, the root will represent a system input, the leaves will represent system outputs, and the intermediate branch nodes will represent internal inputs. Thus, all vertices will be associated with an error permeability value. From the trace trees we find the propagation pathways that errors on system inputs would most likely take by finding the paths from the root to the leaves having the highest weights.

From the permeability graph in Fig. 3 we can now generate the backtrack tree for the system output signal *TOC2*. This tree is shown in Fig. 6.

As illustrated in the backtrack tree (Fig. 6), we have a special relation between the leaves for *ms_slot_nbr* and for *i* and their respective parent. This is because the parent node is also either *ms_slot_nbr* or *i*. Thus, we have an output signal which is connected back to the originating module giving us a recursive relation. In those cases where errors only can enter a system via its main inputs, these branches of the backtrack-trees can be disregarded.

In Fig. 7, we have the trace tree for system input *PACNT*. The trees for inputs *TIC1*, *TCNT*, and *ADC* are very similar to the tree for *PACNT* so they will not be shown here.

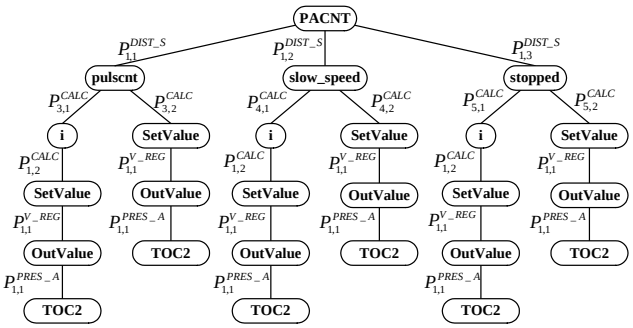

Figure 7. Trace tree for input *PACNT*

As described in Section 4.2, we do not follow the recursion generated by a feedback from a module to itself. In module CALC we have a feedback in signal *i*, and as seen in Fig. 7, we do not have a child node from *i* that is *i* itself.

Here we can see which propagation path from system input to system output has the highest probability. As for backtrack trees, the probability of a path is obtained by multiplying the error permeability values along the path.

4.3 Assessing the Error Exposure of Modules and Signals

To find modules most likely to be exposed to propagating errors, we want to have some knowledge of the "amount" of errors that a module may be subjected to. For this we define the *module error exposure*, X^M, of a module **M** as:

$$0 \leq X^M = \frac{1}{N} \sum \text{incoming arcs of } M \leq 1 \qquad (4)$$

where N is the number of incoming arcs and M is the node in the permeability graph, representing software module **M**. The *module error exposure* is the mean of the weights of all incoming arcs of a node. We define the *non-normalized module error exposure*, $\hat{X}^M$, of a module **M** as:

$$0 \leq \hat{X}^M = \sum \text{incoming arcs of } M \leq N \qquad (5)$$

Going to the signal level, we want to analyse the system and get indications on which signals might be the ones that errors most likely will reach and propagate through. In the backtrack trees we can easily see which error permeability values are directly associated with a signal *s*. We define the set S_p as composed of all unique arcs going to the child nodes of all nodes generated by the signal *s*. A signal may generate multiple nodes in a backtrack tree (see for instance signal *pulscnt* in the backtrack tree in Fig. 6). However, in the set S_p, the permeability values associated with the arcs

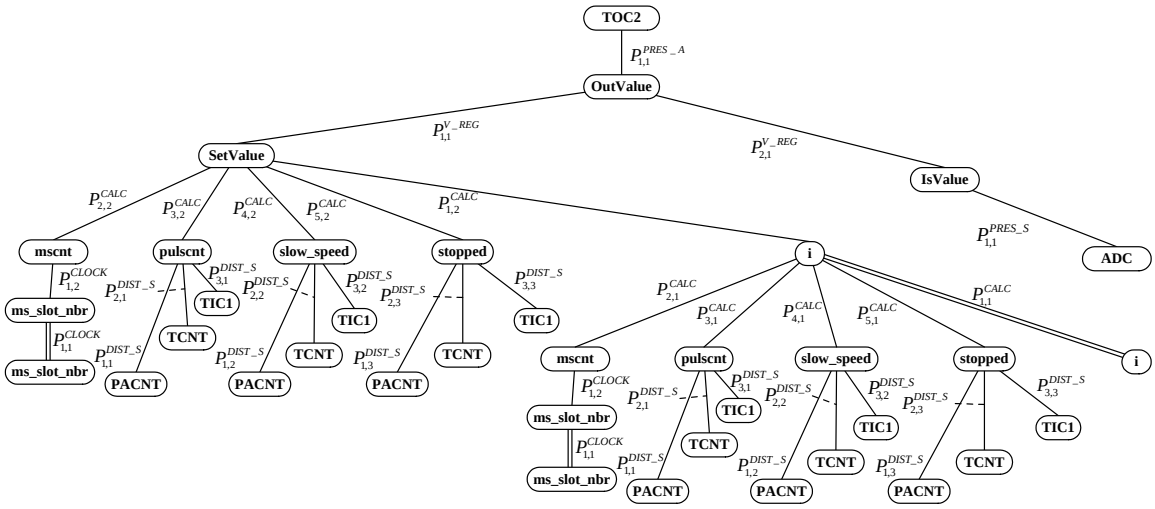

Figure 6. Backtrack tree for output *TOC2*

emanating from those nodes will only be counted once. The *signal error exposure*, X_s^S, of signal *s* is then calculated as:

$$X_s^S = \sum \text{all permeability values in } S_p \quad (6)$$

4.4 Analyzing the Effect of Errors

It may be insufficient to only take into account the propagation characteristics of data errors for a given software system in order to identify weaknesses. Errors that 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 on a system output signal O^{Sys} we must first generate an *impact tree*, which is a generalization of the trace tree described in Section 4.2. Instead of generating a trace tree with a system input as root node, we use the signal of interest in our analysis as the root, in this case s. An impact tree is generated using the same steps as for trace trees (see Fig. 5).

Once we have generated the impact tree for a given signal s, we generate all the propagation paths from the root to the leaves containing system output signal O^{Sys} (there may be leaves which are generated by other system output signals). Each path has a weight associated with it which is the product of all permeability values along that path. We define $s \rightsquigarrow O^{Sys}$, the *impact* of (errors in) s on O^{Sys}, as

$$0 \leq s \rightsquigarrow O^{Sys} = 1 - \prod_i (1 - w_k) \leq 1 \quad (7)$$

where w_k is the weight of path k from s to O^{Sys}. If one could assume independence over all paths, the impact measure would be the conditional probability of an error in s propagating all the way to O^{Sys}. However, as independence can rarely be assumed we will treat this as a relative measure by which different signals can be ranked.

In Eq. 7, the measure only considers one system output signal. If a system has multiple output signals, the corresponding impact value which considers all output signals can be defined as:

$$0 \leq s \rightsquigarrow O^{Sys} = 1 - \prod_i (1 - (s \rightsquigarrow O_i^{Sys})) \leq 1 \quad (8)$$

where $s \rightsquigarrow O_i^{Sys}$ is the impact of signal s on system output signal O_i^{Sys}, i.e., the i^{th} system output signal.

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 output signals, these are not necessarily all equally important for the operation of the system, i.e., some output signals may be more critical than others. For cost-efficiency, one may wish to concentrate resources for dependability on the most critical system output signals and therefore needs to know which signals in the system that are "best" (in a loose sense) to monitor/protect.

Each system output signal O_i^{Sys} is assigned a criticality $C_{O_i^{Sys}}$, 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.

Each signal s has a certain impact, $s \rightsquigarrow O_i^{Sys}$, on sys-

tem output O_i^{Sys}, as calculated according to Eq. 7. The criticality of s as experienced by system output O_i^{Sys}, $C_{s,i}$, is calculated as

$$0 \leq C_{s,i} = C_{O_i^{Sys}} \cdot (s \rightsquigarrow O_i^{Sys}) \leq 1 \quad (9)$$

Once we have the criticality of s with regard to each system output signal O_i^{Sys} we can subsequently compute an overall criticality value. We define the *criticality* C_s of signal s as

$$0 \leq C_s = 1 - \prod_i (1 - C_{O_i^{Sys}} \cdot (s \rightsquigarrow O_i^{Sys})) \leq 1 \quad (10)$$

For each signal, the criticality measure indicates 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 the project policies for software development change.

At this point, we have only defined *impact* and *criticality* at the signal level. If we consider a module **M** in a system with i output signals, we can define the impact of **M** on a given system output signal O_i^{Sys}, $M \rightsquigarrow O_i^{Sys}$, as follows:

$$0 \leq M \rightsquigarrow O_i^{Sys} = 1 - \prod_j (1 - (O_j^M \rightsquigarrow O_i^{Sys})) \leq 1 \quad (11)$$

where $O_j^M \rightsquigarrow O_i^{Sys}$ is the impact of (errors in) the output signal O_j^M of **M** on system output signal O_i^{Sys}. For each output signal of **M**, there is one such impact value. In order to get a measure for the impact of **M** on the system output as a whole we can define $M \rightsquigarrow O^{Sys}$, the *module impact* of **M** on system output, as follows:

$$M \rightsquigarrow O^{Sys} = 1 - \prod_i \prod_j (1 - (O_j^M \rightsquigarrow O_i^{Sys})) \leq 1 \quad (12)$$

Going from impact to criticality is not a big step. Instead of using the individual impact values of the outputs of a module, the corresponding criticality values for the chosen system output signal are used. The criticality of module **M**, with regard to system output O_i^{Sys} can thus be defined as:

$$0 \leq C_i^M = 1 - \prod_j (1 - C_{O_j^M,i}) \leq 1 \quad (13)$$

where $C_{O_j^M,i}$ is the criticality of output O_j^M with regard to system output signal O_i^{Sys}. A total measure regarding all system output signals is then referred to as the *module criticality*, C^M, of **M** and is defined as:

$$0 \leq C^M = 1 - \prod_i \prod_j (1 - C_{O_i^{Sys}} \cdot (O_j^M \rightsquigarrow O_i^{Sys})) \leq 1 \quad (14)$$

4.5 Identifying Hot-Spots and Vulnerabilities

In this section we will discuss how to identify hot-spots and vulnerable locations in the analysed software using the obtained error propagation and error effect profiles.

It is hard to develop a generalized heuristic for identification hot-spots and vulnerabilities. However, the following rules of thumb or recommendations for interpretation of the metrics can be made:

- The higher the error exposure values of a module or signal, the higher the probability that it will be subjected to errors propagating through the system if errors are indeed present.
- The higher the error permeability values of a module or signal, the lower its ability to contain (as in "confine") errors. Thus, there is an increase in the probability of subsequent modules being subjected to propagating errors if errors should pass through the module.
- The higher the criticality (or impact if the system only has one output signal) of a module or signal, the higher the probability of an error there causing damage from a system point-of-view.

When selecting which parts of the software to improve with dependability structures and mechanisms, these rules may not individually yield the same result. Consider the case where a module or signal has a *low exposure* but a *high criticality*. The low exposure means that there is a low probability of errors propagating to that module/signal. However, the high criticality means that, should an error find its way into that module/signal, there is a high probability of that error causing damage which propagates beyond the system barrier into the environment. Thus, one may select modules/signals with low exposure and high criticality for further dependability efforts. For example, a signal with high criticality may be equipped with error detection and recovery mechanisms, or a module with high criticality may be duplicated or triplicated.

From a pure dependability viewpoint, it may be sufficient to only consider the criticality of modules and signals, as these indicate the amount of damage an error may cause. However, from a cost viewpoint, taking care of high criticality events may not be worthwhile if these events have a very low probability of occurring. Using both the propagation and the effect profiles, a cost-benefit analysis can be performed. In this case, project policies will determine whether a low-probability event with high criticality will be taken care of or not.

The obtained profiles may also aid in the design of structures for dependability. For example, a situation with low error exposure and high criticality (impact) indicates that any error detection mechanism in 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 error detection mechanism in that location may suffice.

5 Obtaining Numerical Estimates of Error Permeability

Our method for experimentally estimating the error permeability values of software modules is based on fault injection (FI – see, e.g., [6]).

For analysis of raw experimental data, we make use of so-called Golden Run Comparisons (GRC). A Golden Run (GR) is a trace of the system executing without any injections being made, hence, this trace is used as reference and is stated to be "correct". All traces obtained from the injection runs (IR's, where injections are conducted), are compared to the GR, and any difference indicates that an error has occurred. The main advantage of comparing an injection run with a reference run to detect perturbations is that this does not require any *a priori* knowledge of how the various signals are supposed to behave, which makes this approach less application specific.

Experimentally estimating error permeability is done by injecting errors in the input signals of a module and logging its output signals. We only inject one error in one input signal at a time. Suppose, for module **M**, we inject n_{inj} distinct errors in input *i*, and at output *k* observe n_{err} differences compared to the GR's, then we can directly estimate the error permeability $P^M_{i,k}$ to be $\frac{n_{err}}{n_{inj}}$ (see more on experimental estimation in [1] and [8]).

The type of injected errors and the work load are likely to affect the estimates. Thus, when generating estimates using fault-injection, one should use an error set which resembles real faults and errors as closely as possible, and also use realistic work loads.

5.1 Example Analysis

For estimating error permeability values of our example system, we used the Propagation Analysis Environment (PROPANE [5]). This tool enables fault and error injection, using SWIFI (SoftWare Implemented Fault Injection), in software running on a desktop.

In this study, the aim was to produce an estimate of the *error permeability* of the modules of the target system. As described in Section 5 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.

We injected single bit-flips in the input signals of the modules at 10 different time instances distributed in half-second intervals between 0.5 s and 5.0 s from start of arrestment (although only at one time in each IR). To get a varied load on the system and the modules, we subjected the system to 25 test cases: 5 masses and 5 velocities of the incoming aircraft uniformly distributed between 8,000-20,000 kg, and between 40-80 m/s, respectively.

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.

5.2 Experimental Results and Obtained Profiles

In the target system, we have 25 input/output pairs for which we produced an estimate of the error permeability measure (see Eq. 1) using the method from Section 5. Due to space limitations, we do not include all individual estimates here. However, these values can be found in [4].

In Table 1, we obtain error permeability values (P^M and $\hat{P}^M$, respectively), error exposure values (X^M and $\hat{X}^M$) and impact values ($M \rightsquigarrow TOC2$) for each module.

The modules DIST_S and PRES_S have no error exposure values as they only receive system input signals, i.e., from external sources. This does not mean that these modules will never be exposed to errors on their inputs, but rather that the error exposure is dependent on the probability of errors occurring in the various external data sources. The modules with the highest non-weighted error exposure are the CALC module and the V_REG module. This indicates that these two modules are central in the system and thus candidates for error detection and recovery mechanisms.

The module PRES_A has no impact value since the impact is calculated with regard to its output. One could perhaps say that this module has an impact of 1.0, as an error in its output signal (*TOC2*) is guaranteed to generate an error in the system output signal (also *TOC2*). When calculating module impact, one may also view the environment as a module and calculate its impact on system output. In this case, the system input signals are viewed as the outputs of the environment and calculations are performed as described in Eq. 12. The system only has one output signal. Thus, no criticality values are calculated as these would only be scaled impact values.

In Table 2, we 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).

Table 1. Estimated relative permeability, error exposure and impact values of the modules

Module	P^M	$\hat{P}^M$	X^M	$\hat{X}^M$	$M \rightsquigarrow TOC2$
CLOCK	0.500	1.000	1.000	1.000	0.410
DIST_S	0.107	0.966	-	-	0.698
PRES_S	0.000	0.000	-	-	0.784
CALC	0.299	2.986	0.165	2.473	0.784
V_REG	0.890	1.781	0.247	1.479	0.875
PRES_A	0.875	0.875	0.890	1.781	-

Table 2. Estimated signal error exposures and impacts on *TOC2*

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

Here, an example of how the rules-of-thumb for identification of weaknesses and hot-spots can be in conflict with each other is highlighted. Consider the signal *IsValue* going from PRES_S to V_REG. With the propagation analysis, we obtained a zero error exposure value indicating that errors never (or at least rarely) propagate into this signal. This suggests that *IsValue* may not be a weak part of the software. On the other hand, with the effect analysis, we obtained a very high error impact value. This means that an error in *IsValue* could have a high impact should it occur and may cause severe system failure, which would suggest that *IsValue* may be location which should be considered as a weak spot of the software. Thus, the propagation analysis and the effect analysis may identify different weaknesses of the software and corresponding input to system designers regarding cost/benefit trade-offs and implications of placement and design of structures and mechanisms for dependability.

6 Summary and Conclusions

This paper presents a condensed description of the EPIC framework for analysis of propagation and effect of data errors in software. The system model assumed in this framework is that software is composed of a set of modules interconnected with signals. The results from EPIC can be used for identifying areas (modules and signals) which may prove to be weaknesses and/or hot-spots in the system. Thereby, extra efforts for providing dependability may be directed towards those areas. We also describe a fault-injection based method for obtaining estimates of the metrics and use an example system to illustrate the obtained profiles. Concluding this paper, we state that the EPIC provides means for software profiling which may provide knowledge pertinent to dependability engineering in software systems.

References

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

[2] Fujiwara H., Shimono T. "On the Acceleration of Test Generation Algorithms", *Proc. Int. Symp. on Fault-Tolerant Computing (FTCS-13)*, pp. 98–105, 1983.

[3] 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.

[4] Hiller M., et al., "On the Placement of Software Mechanisms for Detection of Data Errors", *Proc. Int. Conf. on Dependable Systems and Networks (DSN 2002)*, pp. 135–144, 2002.

[5] Hiller M., et al., "PROPANE: An Environment for Examining the Propagation of Errors in Software", *Proc. Int. Symp. on Software Testing and Analysis (ISSTA'02)*, pp. 81–85, 2002.

[6] 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.

[7] Morell L., et al., "Perturbation Analysis of Computer Programs", *Proc. Int. Conf. on Computer Assurance (COMPASS'97)*, pp. 77-87, 1997.

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

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

[10] 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.

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

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

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