EPIC: Profiling the Propagation and Effect of Data Errors in Software

Martin Hiller, *Member*, *IEEE*, Arshad Jhumka, *Student Member*, *IEEE*, and Neeraj Suri, *Senior Member*, *IEEE*

Abstract—We present an approach for analyzing the propagation and effect of data errors in modular software enabling the profiling of the vulnerabilities of software to find 1) the modules and signals most likely exposed to propagating errors and 2) the modules and signals which, when subjected to error, tend to cause more damage than others from a systems operation point-of-view. We discuss how to use the obtained profiles to identify where dependability structures and mechanisms will likely be the most effective, i.e., how to perform a cost-benefit analysis for dependability. A fault-injection-based method for estimation of the various measures is described and the software of a real embedded control system is profiled to show the type of results obtainable by the analysis framework.

Index Terms—Data error propagation, data error effect, software profiling, fault injection, dependability assessment.

1 Introduction

THE advent of computerized consumer products, such as automobiles, mobile systems, etc., has produced a large increase in the need for dependable (or robust) systems. As cost is a relevant issue for such systems, the cost of dependability has to be kept low. Furthermore, as the process of producing multiple copies of software is virtually free compared to producing multiple copies of hardware, the trend is to implement more and more functions in software. Thus, 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 the system is supposed to handle; their nature, frequency, duration, etc. Without knowledge of what kind of threats the system is subject to, it is hard to know how to obtain any dependability. This would make both development as well as assessment/analysis of the system difficult (if not impossible).
2. The available structures and mechanisms for dependability. When developing dependable software, it is important to know the characteristics and properties of the means at one's disposal, including their strengths and weaknesses. The overall architecture of the software is likely to be affected by these properties.
3. The vulnerabilities of the software. In order to be able to incorporate dependability structures and mechanisms where they are the most effective, it is important to know where errors tend to propagate and where errors tend to do the most damage. This will aid in utilizing the available resources where they are likely to be of most use.

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. In this paper, we will show an example of how the obtained software profiles can be used to select locations for error detection mechanisms (EDMs).

Analyzing error propagation and error effects can also complement other analysis activities, for instance, FMECA (Failure Mode Effect and Criticality Analysis). In an iterative development process, profiling the software in each iteration may indicate which modules and signals may require more attention during that iteration. Thus, error propagation and effect analysis, as a means of both system analysis and resource management, may be a very useful development-stage tool in such systems.

The focus of *EPIC* is on handling data errors and we consider logically distributed software functions 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

- *M. Hiller is with the Department of Electronics and Software, Volvo Technology Corporation, Sven Hultins Gata 9C, SE-41288 Gothenburg, Sweden. E-mail: martin.hiller@volvo.com.*
- *A. Jhumka and N. Suri are with the Department of Computer Science, TU Darmstadt, Wilhelminenstr. 7, DE-64283 Darmstadt, Germany. E-mail: {arshad, suri}@informatik.tu-darmstadt.de.*

Manuscript received 5 Mar. 2002; revised 17 Mar. 2003; accepted 27 Aug. 2003.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 116022.

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 resemble those used for reliability diagrams and fault trees.

EPIC provides a consolidated and comprehensive software profiling framework building upon and extending the individual framework components introduced in [12], [13].

Paper organization. Section 2 reviews related work and Section 3 describes the assumed system model. The EPIC framework is described in Section 4 and methods for estimating numerical values of the introduced metrics are discussed in Section 5. In Section 6, we illustrate the framework on an example target system and produce estimates for the various metrics. In Section 7, we use the obtained profiles to guide the selection of locations for error detection mechanisms called Executable Assertions, which are then evaluated with regard to detection coverage in Section 8. Some limitations and caveats of the proposed framework are discussed in Section 9. Finally, Section 10 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 [26], the PODEM-algorithm [9], and the FAN-algorithm [7] (which improves on the PODEM-algorithm).

In [30], the authors analyze fault/error propagation and data flow diagrams in VHDL-models in order to select a fault list for fault injection which will fully exercise the fault detection and processing aspects of the model.

Error propagation in hardware is also addressed in [29], where a stochastic propagation model based on error propagation times is described. However, the authors do not target identification of locations for dependability structures and mechanisms as is done in this paper. Also, the model is defined at the module level, i.e., if there are several signals linking two modules together, these will not be considered individually, but as a group.

An approach for dependability analysis, including error propagation, based on data flow analysis in HW-SW codesign is presented in [4]. Here, a data flow model of the system (including only functional requirements) is extended with information regarding fault occurrence, fault latency, and detection probabilities such that a dependability analysis can be performed. This approach works on a high-level model of a system which is not yet divided into hardware and software.

Propagation analysis in software has been described for debugging use in [33]. 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 [22], [34] for analyzing source code under test in order to determine test cases that would reveal the largest amount of defects. In [35], 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.

A method based on control flow analysis is described in [8]. Here, a software system is analyzed with regard to control flow and, based on the results of this analysis, flow checks are placed in order to detect errors dynamically. As this approach only deals with control flow errors, it is very different from ours as we deal with data errors. The control flow approach will not handle detection of data errors unless these change the control flow such that it can be detected by the obtained detectors.

An investigation in [21] reported that there was evidence of uniform propagation of data errors. That is, a data error occurring at a location l in a program would, to a high degree, exhibit uniform propagation, meaning that, for location l, either all data errors would propagate to the system output or none of them would.

Finding optimal combinations of hardware mechanisms for error detection based on experimental results was described in [31]. The authors used coverage and latency estimates for a given set of error detection mechanisms (EDMs) to form subsets which minimized overlapping between different EDMs, thereby giving the best cost-performance ratio.

In [18], a study on the use of self-checks and voting for software error detection concludes, among other things, that placement of self-checks seemed to cause problems, i.e., self-checks that might have been effective failed on account of being badly placed. In our work, we provide means for a rigorous placement process.

3 Software and 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 software module performs computations using the provided inputs to generate the outputs. At the lowest level, such a black-box module may be a procedure or a function but could also conceptually be a basic block or particular code fragment within a procedure or function (at a finer level of software abstraction). A number of such modules constitute a system and they are inter-linked via signals. Of course, this system may be seen as a larger component or module in an even larger system. Signals can originate internally from a module, e.g., as a calculation result, or externally from the hardware itself, e.g., a sensor reading from a register. The destination of a signal may also be internal, being part of the input set of a module, or external, for example, the value placed in a hardware register.

Software constructed as such is found in numerous embedded systems. For example, most applications controlling physical events such as in automotive systems are

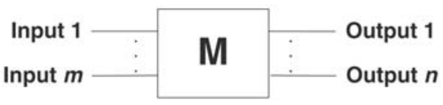


Fig. 1. A basic black-box software module with m inputs and n outputs.

traditionally built up as such. 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 arresting aircraft and is described in Section 6.

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.

4 EPIC: Generating Software Profiles

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: 1) error propagation profile and 2) 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.

A data error has a probability of affecting the system such that further errors are generated during operation. If one could obtain knowledge of the error propagation characteristics of a particular system, this would aid the development of techniques and mechanisms for detecting and eventually correcting the error.

Such knowledge can translate into improved effectiveness of dependability structures and mechanisms and the consequent cost/performance-ratio of these, as the efforts can be concentrated to where errors tend to propagate and/or do the most damage. In particular, if an erroneous signal or module has a very detrimental effect on system output, knowing how errors propagate from that signal or module will help in using resources for dependability in an efficient way.

The results obtained using EPIC are useful even with minimal knowledge of the distribution of the occurring data errors (within our assumed fault model), i.e., if one does not know which errors are most likely to appear, since we are only interested in a partial ordering which is achieved by profiling. Knowledge on error distribution will surely improve the value of the results, but performing the analysis without it still provides qualitative insights on system error susceptibility.

4.1 Error Permeability—Letting Errors Pass

In our approach, we introduce the measure of *error permeability* and, based on it, we define a set of related measures that cumulatively provide an insight on the error propagation and effect characteristics and vulnerabilities of a system.

Consider the software module in Fig. 1 (at this point only discrete software modules are considered). Starting with a simple definition of error permeability, refinements will follow successively. 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 $\mathbf{M}$, the *error permeability*, $P^M_{i,k}$, is defined as follows:

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

This measure indicates how *permeable* an input/output pair of a software module is to errors occurring on that particular input. One characteristic of this definition of error permeability is that it is independent of the probability of error occurrence on the input (as it is a conditional probability). On the other hand, error permeability is still dependent on the workload of the module as well as the type of the errors that can occur on the inputs. 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.

Error permeability is defined at the signal level, i.e., an error permeability value characterizes the error propagation from one input signal to one output signal in a given module. Going to the module level (Fig. 1), we define the *module error permeability*, P^M, of a module $\mathbf{M}$ with m input signals and n output signals, to be:

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

Note that this does not necessarily reflect the overall probability that an error is permeated from the input of the module to the output. Rather, it is an abstract measure that can be used to obtain a relative ordering across modules. If all inputs are assumed to be independent of each other and errors on one input signal can only generate errors on one output signal at a time, then this measure is the actual probability of an error on the input permeating to the output. However, this is seldom the case in most practical applications.

One potential limitation of this measure is that it is not possible to distinguish modules with a large I/O count (i.e., a large number of input and output signals) from those with a small I/O count. This distinction is useful to ascertain as modules with large I/O count are likely to be central parts (almost like hubs) of the system, thereby attracting errors from different parts of the system. In order to be able to make this distinction, we remove the weighting factor in (2), thereby, in a sense, "punishing" modules with a large I/O count. Thus, for a module $\mathbf{M}$ with m input signals and n output signals, we can define the *nonnormalized module error permeability*, $\hat{P}^M$ as follows:

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

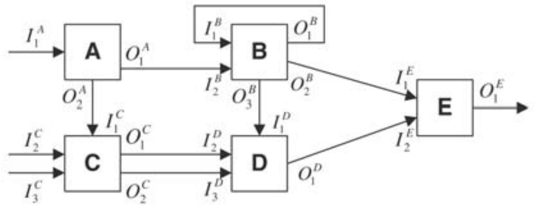


Fig. 2. A 5-module example SW system.

Similarly to the module error permeability, this measure does not have a straightforward real-world interpretation but is a measure that can be used during development to obtain a relative ordering across modules. The larger this value is for a particular module, the more effort has to be spent in order to increase the error containment capability of that module (which is the same as decreasing the error permeability of the module), for instance, by using wrappers as in [28]. Note that, as the maximum value of each individual permeability value is 1, the upper bound for this measure is the product of the number of inputs (m) and outputs (n).

The two measures defined in (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

So far, we have obtained error permeability factors for each discrete software module in a system. Considering every module individually does have limitations; this analysis will give insights on which modules are likely (relatively) to transfer incoming errors, but will not reveal modules likely to be exposed to propagating errors in the system. 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. 2. Here, we have five modules, **A** through **E**, connected to each other with a number of signals. The ith input of module **M** is designated I_i^M and the kth output of module **M** is designated O_k^M. External input to the system is received at I_1^A, I_2^C, and I_3^C. The output produced by the system is O_1^E.

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. Each node in the graph corresponds to a particular module and has a number of incoming arcs and a number of outgoing arcs. Each arc has a weight associated with it, namely, the error permeability value. Hence, there may be more arcs between two nodes than there are signals between the corresponding modules. Actually, the maximum number of outgoing arcs for a node is the product of the number of incoming signals and the number of outgoing signals for the corresponding software module (each input/output pair of a module has an error permeability value). Arcs with a zero weight (representing nonpermeability from an input to an output) can be omitted. With the permeability graph, we can:

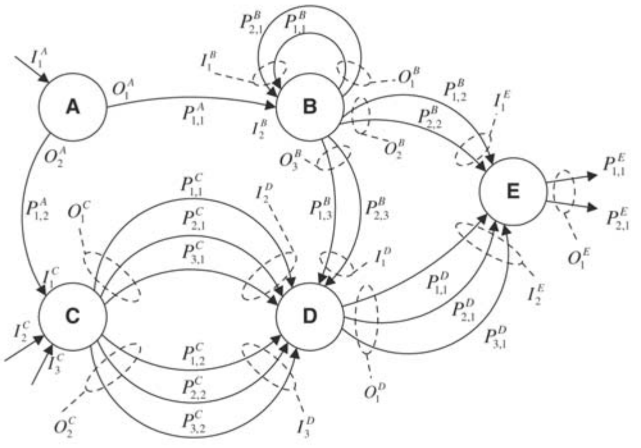


Fig. 3. Permeability graph for Fig. 2.

- 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
- Trace errors from system input signals to system output signals in order to find which path 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. These backtrack trees can be constructed quite simply based on the following steps on the permeability graph, namely:

- **Step A1**. Select a system output signal and let it be the root node of the backtrack tree.
- **Step A2**. For each error permeability value associated with the signal, generate a child node that will be associated with an input signal.
- **Step 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 subtree for the child node from Step 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.
- **Step A4**. If there are more system output signals, go back to Step A1.

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.

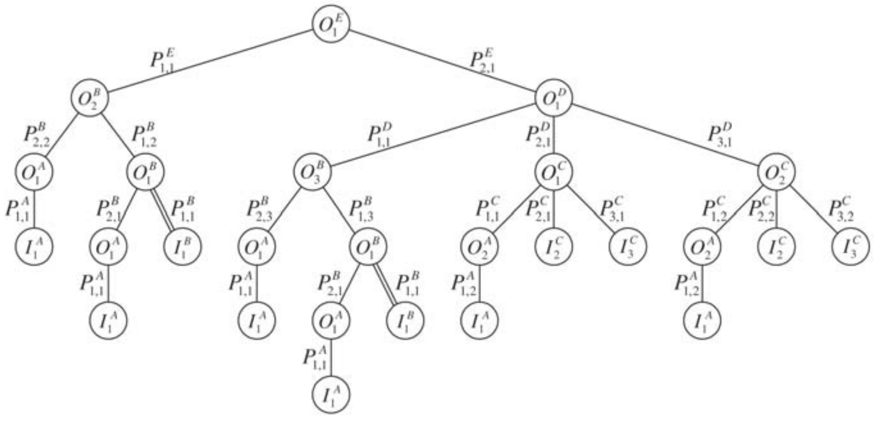


Fig. 4. Backtrack tree of system output O_1^E of example system.

Input error tracing is achieved similarly. However, instead of constructing a backtrack tree for each system output, we construct a *trace tree* for each system input, as follows:

- **Step B1**. Select a system input signal and let it be the root node of the trace tree.
- **Step 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.
- **Step 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 subtree of the child node from Step 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 subtrees for the remaining outputs. We do not follow the recursion generated by this feedback.
- **Step B4**. If there are more system input signals, go back to Step B1.

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.

The case when an output of a module is connected to an input of the same module is handled in the way described in Step A3 of the backtrack tree generation script. If we use recursive subtree generation we would get an infinite number of subtrees with diminishing probabilities. As all permeability values are ≤ 1, the subtree with the highest probability is the one which only goes one pass through the feedback loop and this path is included in the permeability tree. In [7], [9], [26], similar techniques are used for hardware error propagation analysis.

The backtrack tree for system output O_1^E of the example system is shown in Fig. 4. Here, we observe the double line between I_1^B and O_1^B. This notation implies that we have a local feedback in module $\mathbf{B}$ (O_1^B is connected to I_1^B) and represents breaking up of the propagation recursion.

The weight for each path is the product of the error permeability values along the path. For example, in Fig. 4, the path from O_1^E to I_1^A going straight from O_1^A (connected to I_2^B) to O_2^B (the leftmost path in the tree) has the probability $P = P_{1,1}^A \cdot P_{2,2}^B \cdot P_{1,1}^E$. This is the conditional probability that, given an error in O_1^E and the error originated from I_1^A, it propagated directly through O_2^B which is connected to I_1^E and then to O_1^E.

If we have knowledge regarding the probability of errors appearing on the input signals, we can use these probabilities as additional weights on the paths. For example, if the probability of an error appearing on I_1^A is $Pr(I_1^A)$, then the P can be adjusted with this factor, giving us $P' = Pr(I_1^A) \cdot P_{1,1}^A \cdot P_{2,2}^B \cdot P_{1,1}^E$. This is the probability of an error appearing on system input I_1^A, propagating through module $\mathbf{B}$ directly via O_2^B to system output O_1^E.

The trace tree for system input I_1^A is shown in Fig. 5. 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.

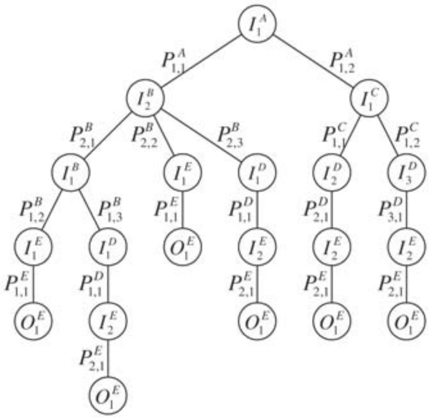


Fig. 5. Trace tree for system input I_1^A of example system.

For example, in Fig. 5, the probability of an error in I_1^A propagating to module C and via its output O_2^C to module D and from there via module E to system output O_1^E is $P = P_{1,2}^A \cdot P_{1,2}^C \cdot P_{3,1}^D \cdot P_{2,1}^E$. Again, if we know that $Pr(I_1^A)$ is the probability of an error appearing on I_1^A, then we can adjust P to get $P' = Pr(I_1^A) \cdot P_{1,2}^A \cdot P_{1,2}^C \cdot P_{3,1}^D \cdot P_{2,1}^E$.

4.3 Assessing the Error Exposure of Modules and Signals

Using the backtrack and trace trees enables determining two specific aspects: 1) the paths in the system that errors will most likely propagate along to get to certain output signals and 2) which output signals are most likely affected by errors occurring on the input signals. With this knowledge, we can start building a propagation profile of the software in order to identify its weaknesses.

However, once we have the most probable propagation paths, we still have to find the modules along that path that are the best (in some sense) to target for dependability efforts. Earlier, in (2) and (3), we had defined two measures, *module error permeability* and *nonnormalized module error permeability*, that can guide us in this search.

These measures only consider the permeability values of discrete modules—couplings across modules are disregarded. Using the permeability graph, we now define a set of measures that explicitly consider coupling and aid determining the weak spots and vulnerabilities of the software. 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 $\mathbf{M}$ as:

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

where N is the number of incoming arcs and M is the node in the permeability graph, representing software module $\mathbf{M}$. This measure does not consider any correlation that may exist between two or more incoming arcs. Remember that we do profiling to get a partial ordering and, thus, this is not a hindrance for us. The *module error exposure* is the mean of the weights of all incoming arcs of a node. Analogous to the *nonnormalized module error permeability*, we can also define the *nonnormalized module error exposure*, $\hat{X}^M$, of a module $\mathbf{M}$ as:

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

This measure does not have a real-world interpretation either—it is used only during system analysis to obtain a relative ordering between modules. The two exposure measures ((4) and (5)) along with the previously defined permeability measures ((2) and (3)) will be the basis for the analysis performed to obtain a profile showing the error propagation characteristics of the software. As was the case for the two module permeability measures, the two exposure measures, *module error exposure* and *nonnormalized module error exposure*, are used for distinguishing between nodes with a small number of incoming arcs and those with a large number.

The error exposure measures defined in (4) and (5) indicate which modules will most probably be the ones exposed to errors propagating through the system. If we want to analyze the system at the signal level and get indications on which signals might be the ones that errors most likely will reach and propagate through, we can define a measure which is analogous to the error exposure defined in (4), but is only calculated for one signal at a time. 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 O_1^B in the backtrack tree in Fig. 4). However, in the set S_p, the permeability values associated with the arcs emanating from those nodes will only be counted once. The *signal error exposure*, X_s^S, of signal s is then calculated as:

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

The interpretation for the signal error exposure is analogous to the module error exposure. That is, the higher a signal error exposure value, the higher the probability of errors in the system being propagated through that signal.

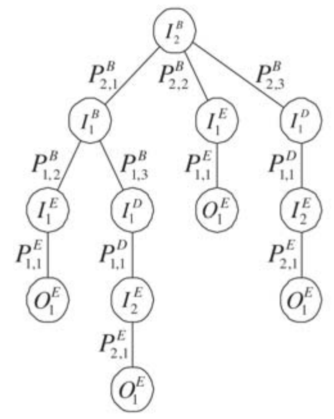
We have now defined a basic analytical framework for ascertaining measures pertaining to error propagation and software vulnerability. In the following sections, we augment the framework with measures for analyzing the effect of errors on the final output of the system as well as for obtaining a measure of criticality of signals. The knowledge gained in the propagation analysis combined with the knowledge gained in the effect analysis will help in profiling the software such that weaknesses and vulnerabilities can be identified.

4.4 Analyzing the Effect of Errors on System Output

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 analyze to what extent errors in a signal (system input signal or intermediate signal) affect the system output, i.e., what the *impact* of errors on the system output signals is.

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. The steps for generating an impact tree are otherwise the same as for generating a trace tree.

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 four paths generated by the impact tree:

$P^B_{2,1}P^B_{1,2}P^E_{1,1} = w_1$

$P^B_{2,1}P^B_{1,3}P^D_{1,1}P^E_{2,1} = w_2$

$P^B_{2,2}P^E_{1,1} = w_3$

$P^B_{2,3}P^D_{1,1}P^E_{2,1} = w_4$

Fig. 6. Impact tree for intermediate signal I^B_2.

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:

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

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. 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 deciding where dependability structures and mechanisms have to be incorporated, one may consider 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, should they occur, are likely to be costly).

In (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:

$$0 \leq s \rightsquigarrow O^{Sys} = 1 - \prod_i (1 - (s \rightsquigarrow O^{Sys}_i)) \leq 1, \tag{8}$$

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

To further illustrate the concept of impact, again consider the example shown in Fig. 2. Suppose that we would like to calculate the impact of errors in signal I^B_2 on system output O^E_1. First, we will generate an impact tree, as shown in in Fig. 6.

The impact tree shown in Fig. 6 is actually the left subtree of the trace tree for system input signal I^A_1 shown in Fig. 5. In order to calculate the impact of errors in I^B_2 on system output O^E_1, we generate all the propagation paths from the root to the leaves. In this case, with only one system output, all leaves are considered. This gives us four paths, as shown in Fig. 6. Using the weights of the paths, we can now calculate $I^B_2 \rightsquigarrow O^E_1$, i.e., the impact of (errors in) I^B_2 on O^E_1, as:

$$\begin{aligned} I^B_2 \rightsquigarrow O^E_1 &= 1 - \prod_{i=1}^{4} (1 - w_i) \\ &= 1 - (1 - w_1)(1 - w_2)(1 - w_3)(1 - w_4), \end{aligned}$$

where w_i are the weights listed in Fig. 6.

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^{Sys}_i is assigned a criticality $C_{O^{Sys}_i}$, 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 has a certain impact, $s \rightsquigarrow O^{Sys}_i$, on system output O^{Sys}_i, as calculated according to (7). The criticality of s as experienced by system output O^{Sys}_i, $C_{s,i}$, is calculated as:

$$0 \leq C_{s,i} = C_{O^{Sys}_i} \cdot (s \rightsquigarrow O^{Sys}_i) \leq 1. \tag{9}$$

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

$$\begin{aligned} 0 \leq C_s &= 1 - \prod_i (1 - C_{s,i}) \\ &= 1 - \prod_i (1 - C_{O^{Sys}_i} \cdot (s \rightsquigarrow O^{Sys}_i)) \leq 1. \end{aligned} \tag{10}$$

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.

Note that, if the system only has one output signal, then the obtained criticality will only function as a constant scaling factor of the impact values, i.e., the relative order among the signals of the system will not change. Thus, calculating criticality values is essential only when there are multiple output signals in a system and these are of different "importance." At this point, we have only defined *impact* and *criticality* at the signal level. Going up in abstraction levels, we can now define equivalent measures which are based on the signal level measures, but consider entire modules instead. If we consider a module $\mathbf{M}$ in a system with i output signals, we can define the impact of $\mathbf{M}$ on a given system output signal O_i^{Sys}, $M \rightsquigarrow O_i^{Sys}$, as follows:

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

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

$$\begin{aligned} M \rightsquigarrow O^{Sys} &= 1 - \prod_i (1 - (M \rightsquigarrow O_i^{Sys})) \\ &= 1 - \prod_i (1 - (1 - \prod_j (1 - (O_j^M \rightsquigarrow O_i^{Sys})))) \\ &= 1 - \prod_i \prod_j (1 - (O_j^M \rightsquigarrow O_i^{Sys})) \leq 1. \end{aligned} \tag{12}$$

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 $\mathbf{M}$, with regard to system output O_i^{Sys} can thus be defined as:

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

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 $\mathbf{M}$ and is defined as:

$$\begin{aligned} 0 \leq C^M &= 1 - \prod_j (1 - C_{O_j^M}) \\ &= 1 - \prod_j (1 - (1 - \prod_i (1 - C_{O_j^M, i}))) \\ &= 1 - \prod_j (1 - (1 - \prod_i (1 - C_{O_i^{Sys}} \cdot (O_j^M \rightsquigarrow S_{o,i})))) \\ &= 1 - \prod_i \prod_j (1 - C_{O_i^{Sys}} \cdot (O_j^M \rightsquigarrow O_i^{Sys})) \leq 1. \end{aligned} \tag{14}$$

Note that the difference between impact and criticality (see (8) and (10) for signals and (12) and (14) for modules) is the criticality factor $C_{O_i^{Sys}}$ of system output signal O_i^{Sys} which is defined by the system designer. Thus, criticality is a biased version of impact.

We have now defined a number of measures for analyzing the propagation of errors and the effect of errors on system output. In the following section, the introduced measures and their interpretations are summerized.

4.5 Identifying Hot-Spots and Vulnerabilities

The EPIC framework introduced in Section 4 contains a number of metrics for analyzing the propagation and effect of errors in software, namely:

- error permeability for input/output pairs ($P_{i,k}^M$, (1)),
- error permeability for modules (normalized, P^M, and nonnormalized, $\hat{P}^M$, (2) and (3), respectively),
- error exposure for modules (normalized, X^M, and nonnormalized, $\hat{X}^M$, (4) and (5), respectively),
- error exposure for signals (X_s^S, (6)),
- impact for signals ($s \rightsquigarrow O^{Sys}$, (7) and (8)),
- criticality for signals (C_s, (10)),
- impact for modules ($M \rightsquigarrow O^{Sys}$, (12)) and, finally,
- criticality for modules (C^M, (14)).

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

4.5.1 *Interpreting the Metrics*

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, 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 focus error detection efforts in those modules than in those with lower error exposure. An analogous way of reasoning is valid also for the signal error exposure.
- The higher the error permeability values of a module, 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. Therefore, it may be more cost effective to increase the error confinement abilities of those modules than of those with lower error permeability.
- 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 error detection and recovery mechanisms which have the highest criticality (impact). An analogous way of reasoning is valid also for the module criticality (impact).

4.5.2 *Trade Offs and Project Policies*

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.

One way of having a more manageable approach in a project may be to set up certain conditions which must be met by the software. 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 and/or error exposure values. Thus, if a module or signal exceeds these limits, this indicates that more resources have to be allocated to that module to increase its error handling capabilities. A similar approach can be used for criticality (or impact).

The obtained software 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.

Next, we describe how to obtain experimental estimates of the measures and use of our framework on actual software of an embedded control system.

5 Obtaining Numerical Estimates of Error Permeability

Obtaining numerical values for the error permeability may prove to be quite difficult, given that many factors, such as error type, operational profiles, etc., have to be taken into account. This may render it impossible to get the "real" value of the error permeability values of a software system. Thus, a method of estimating these values is needed. In this section, we describe an experimental method based on fault/error injection for obtaining estimates of error permeability values. However, other approaches, such as data flow analysis and other static compiler-assisted approaches, might be investigated in the future.

Our method for experimentally estimating the error permeability values of software modules is based on fault injection (FI). FI artificially introduces faults and/or errors into a system and has been used for evaluation and assessment of dependability for several years (see, e.g., [1], [2], [6]). A comprehensive survey of experimental analysis of dependability appears in [15].

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 (IRs, 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 values for error permeability of a module is done by injecting errors in the input signals of the module and logging its output signals. We only inject one error in one input signal at a time. Suppose, for module $\mathbf{M}$, we inject n_{inj} distinct errors in input i and at output k observe n_{err} differences compared to the GRs, 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 [5] and [23]).

Since the propagation of errors may differ based on the system workload, it is generally preferred to have realistic input distributions over randomly generated inputs. This generates error permeability estimates that are closer to the "real" values.

The type of injected errors may also affect the estimates. Recall that, in order to develop dependable software, one needs to know what type of faults and errors the software shall be able to handle. When generating estimates of the permeability values of the software, one should use an error set which resembles these faults and errors as closely as possible (choosing sets of artificial faults and errors for fault injection which resemble real faults and errors is a large area of research in itself, see, e.g., [10], [16], [20]).

6 Experimental Analysis: An Example Embedded System

In order to illustrate the proposed methodology of profiling error propagation and effect in software, we have conducted an example study on an embedded control system. This study illustrates the results obtained using the EPIC framework and experimental estimates for error permeability values.

6.1 Target Software System

The target system is an embedded control system used for arresting aircraft on short runways and aircraft carriers. The system aids incoming aircraft to reduce their velocity, eventually bringing them to a complete stop. The system is constructed according to specifications found in [32]. The system is illustrated in Fig. 7.

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 nonpreemptive. Thus, from the software viewpoint, there is no difference in running on the actual hardware or running on a desktop computer. Glue software was developed to simulate registers for A/D-conversion, timers, counter registers, etc., accessed by the application. An environment simulator used in experiments conducted on the real system was also ported, so the environment experienced by the real system and the desktop system was identical. The simulator handles the rotating drum and the incoming aircraft (see Fig. 8).

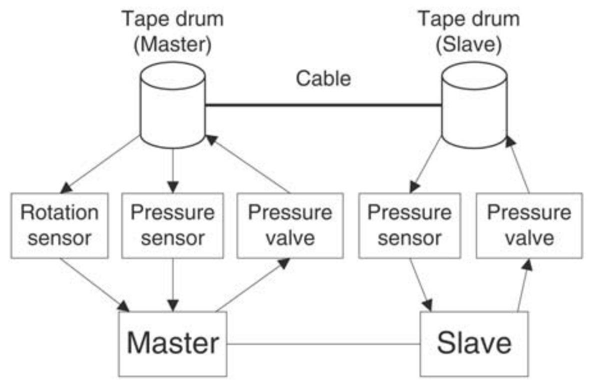


Fig. 7. Target used in example study.

In the real system, there are two nodes: a master node calculating the desired pressure to be applied and a slave node receiving the desired pressure from the master. Each node controls one of the rotating drums. In our setup, the slave was removed and the retracting force applied by the master was also applied on the slave-end of the cable.

The structure of the software is illustrated in Fig. 9. 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 system operates in seven 1ms-slots. In each slot, one or more modules (except for CALC) are invoked. The signal *ms_slot_nbr* tells the module scheduler the current execution slot. Period = 1 ms.

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*, i.e., if the velocity is below a certain threshold or if it has stopped. Period = 1 ms.

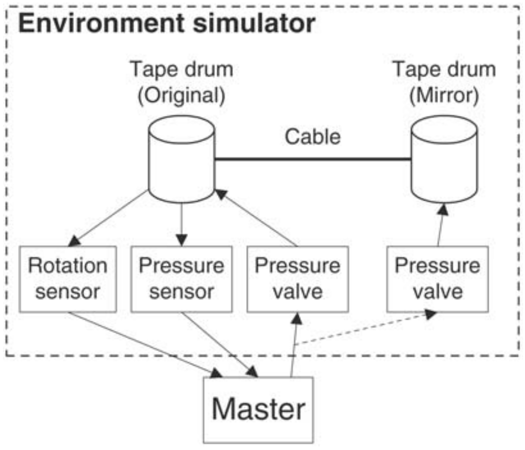


Fig. 8. Target and environment simulator.

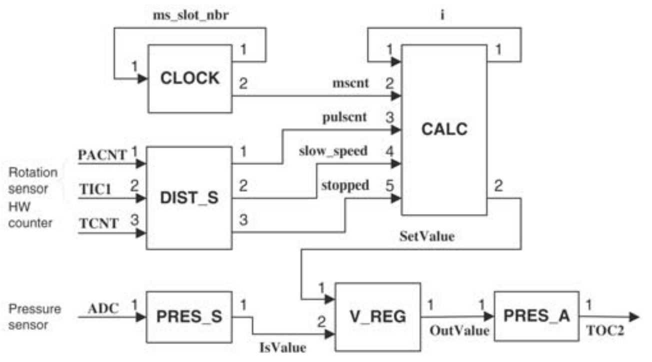


Fig. 9. Software structure of target.

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 predefined *pulscnt*-values corresponding to the various checkpoints. The current checkpoint is stored in i. Period = n/a (background task, runs when other modules are dormant).

PRES_S reads 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*. Period = 7 ms.

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*. This module contains a software-implemented PID-regulator. Period = 7 ms.

PRES_A uses *OutValue* to set the pressure valve via the hardware register *TOC2*. Period = 7 ms.

6.2 System Analysis

Prior to running the experiments, we generated the permeability graph and the backtrack trees and trace trees for the target system as per the process described in Section 4. The permeability graph is shown in Fig. 10.

In the graph (Fig. 10), 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. 9. For instance, $P_{2,1}^{CALC}$ is the error permeability from input 2 (*mscnt*) to output 1 (i) of module CALC. From the permeability graph in Fig. 10, we can now generate the backtrack tree for the system output signal *TOC2*, using the steps described in Section 4.2. This tree is shown in Fig. 11.

As illustrated in the backtrack tree (Fig. 11), 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.

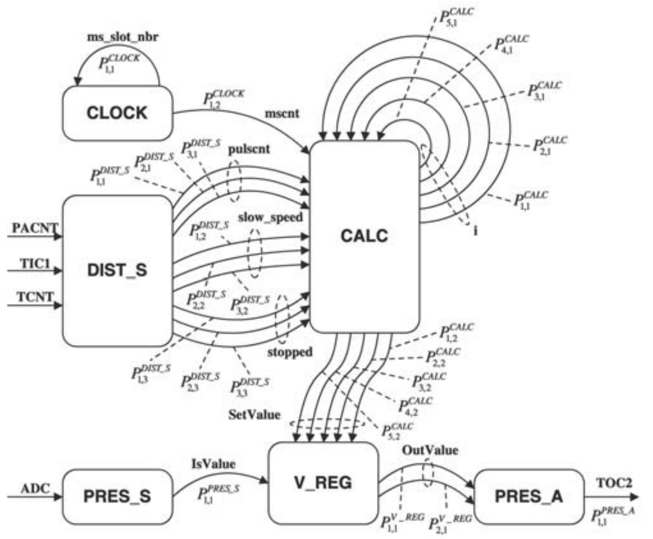


Fig. 10. Permeability graph of target.

In Figs. 12 and 13, we have the trace trees for system input *ADC* and system input *PACNT*, respectively, as obtained by the processes defined in Section 4.2. The trees for inputs *TIC1* and *TCNT* are very similar to the tree for *PACNT* so they will not be shown here.

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 can be seen in Fig. 13, we do not have a child node from i that is i itself.

In order to calculate impact values for the various signals, we generated their respective impact trees. Depicted in Fig. 14, we have the impact tree of the signal *pulscnt* (other impact trees have been left out, but are easily generated by the interested reader).

The impact tree shown in Fig. 14 is actually the left subtree of the trace tree for system input signal *PACNT*

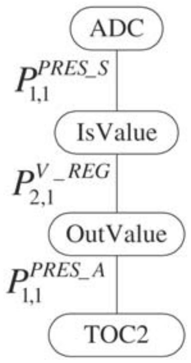


Fig. 12. Trace tree for input *ADC*.

shown in Fig. 13. 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. Here, with only one system output, all leaves are considered.

At this point, we have generated all trees and graphs required for our analysis of the software. The next step is to estimate numerical values for the individual permeability values such that we can use the analysis results to identify the modules and signals, which may prove to be weaknesses or hot-spots in the system.

6.3 Experimental Setup

For estimating error permeability values, we used the Propagation Analysis Environment (PROPANE [14]). This tool enables fault and error injection, using SWIFI (SoftWare Implemented Fault Injection), in software running on a desktop (currently for the Win32 platform). The tool is also capable of creating traces of individual variables and different predefined events during the execution. Each trace of a variable from an injection experiment is compared to the corresponding trace in the Golden Run. Any discrepancy is recorded as an error.

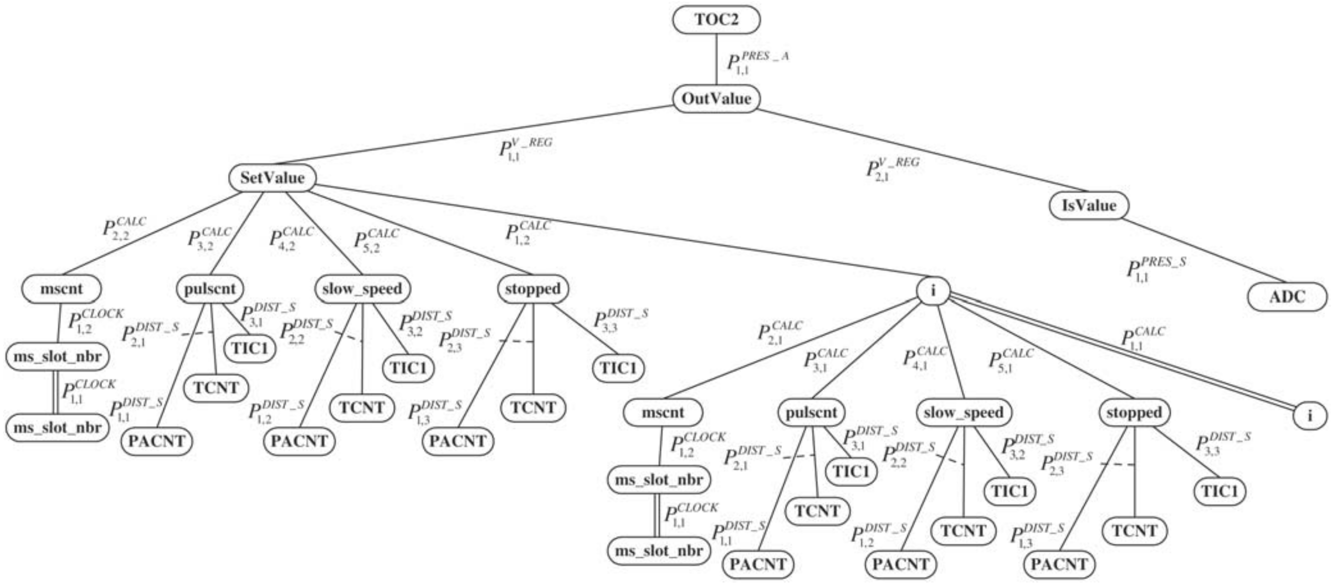


Fig. 11. Backtrack tree for output *TOC2*.

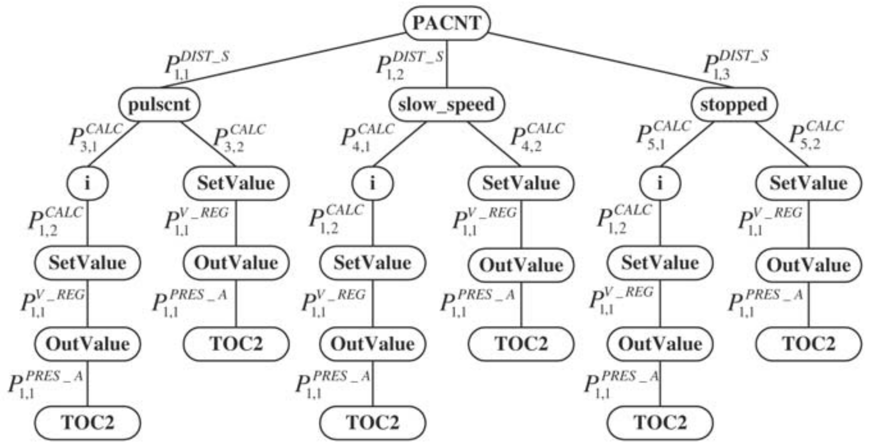


Fig. 13. Trace tree for input *PACNT*.

For logging and injection, the target system was instrumented with high-level software traps. As a trap is reached during execution, an error is injected and/or data logged. The traces obtained during execution have millisecond resolution for every logged variable. Also, we ported the software to run on a desktop system, so the intrusion of the traps is nonexistent in our setup as it runs in simulated time.

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.

Let us assume that we wish the software to be able to handle single event upsets (SEUs) caused by, e.g., radiation. Thus, when producing the estimated error permeability values, we use single bit-flips as our error model as these have been found to emulate SEUs well [25]. The input signals are all 16-bit wide, except *PACNT*, which is 8-bit wide. We injected bit-flips in each bit position (one position at a time) at 10 different time instances distributed in half-second intervals between 0.5s and 5.0s from start of arrestment (although only at one time in each IR). In order to get a varied load on the system and the modules, we subjected the system to 25 test cases: five masses and five velocities of the incoming aircraft uniformly distributed between 8,000-20,000kg and between 40-80m/s, respectively. Thus, for each input signal, we conducted $16 \cdot 10 \cdot 25 = 4,000$ injections (2,000 for *PACNT*).

The raw data obtained in the IRs 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. In our experimental setup—real software running in simulated time, in a simulated environment, and on simulated hardware—this is a valid way of comparing traces even for continuous signals where fluctuations between similar runs in a real environment may be normal.

When performing the GRC, we only considered direct propagation. That is, once errors in an IR had propagated beyond the system boundary (via the system output signal) and affected the environment such that the inputs to the system change (compared to the corresponding GR), the GRC was stopped.

6.4 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 (1)) using the method from Section 5. These values (Table 1) form the basis for subsequent results, which are calculated as described in Section 4.

In Table 2, we obtain normalized and nonnormalized module error permeability values (P^M and $\hat{P}^M$, respectively), normalized and nonnormalized module error exposure values (X^M and $\hat{X}^M$) and module 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 nonweighted error exposure are the CALC module and the V_REG module. This indicates that these two modules are central in the

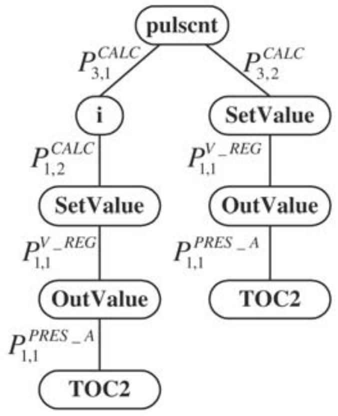


Fig. 14. Impact tree for signal *pulscnt*.

TABLE 1
Estimated Error Permeability Values of the Input/Output Pairs

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

system and that they are good 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 (12). The system only has one output signal. Thus, no criticality values are calculated as these would only be scaled impact values.

From the backtrack tree in Fig. 11, we can generate 22 propagation paths from the system output signal to an input signal. Each of these paths has a total weight, which is the product of the permeability values of the arcs in the path. Ordering the paths according to their total weight gives us some knowledge of the more probable paths for error propagation. Table 3 depicts the three paths that acquired weights greater than zero (the paths along which errors might propagate).

In Table 4, 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).

The same information as in Table 4 is depicted graphically in Figs. 15 and 16. 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 (because it is either a system input or a system output).

In Figs. 15 and 16, 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 (see Fig. 15) 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

TABLE 2
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 3
The Three Nonzero Propagation Paths and Their Weights

Path/Product	Weight
$P^{CALC}_{1,1} P^{CALC}_{1,2} P^{V\_REG}_{1,1} P^{PRES\_A}_{1,1}$	0.04337
$P^{DIST\_S}_{1,1} P^{CALC}_{3,1} P^{CALC}_{1,2} P^{V\_REG}_{1,1} P^{PRES\_A}_{1,1}$	0.02050
$P^{DIST\_S}_{1,2} P^{CALC}_{4,2} P^{V\_REG}_{1,1} P^{PRES\_A}_{1,1}$	0.00691

TABLE 4
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

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 a 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.

Next, we show a small example of how the information provided by the obtained profiles can be used in the development of dependable software, more specifically, in the placement of error detection mechanisms (EDMs).

7 Selecting Locations for EDMs: An Example of How to Use the Obtained Profiles

In this section, we will select locations for error detection mechanisms (EDMs) based on the profiles obtained in the analysis of our target system. We will select two sets of locations, one based on the propagation profile only and one based on the propagation and effect profiles.

The mechanisms we have chosen to use for this study are so called Executable Assertions (henceforth, called EAs) and are commonly used in embedded software (see, e.g., [19], [24], [27]). EAs are usually small snippets of code which are executed online to check that certain constraints on the values of variables are not violated, such as minimum and maximum values and change rate limitations. The specific EAs used in this paper are generic parameterized mechanisms aimed at individual signals and are described in [11]. The mechanisms we have at our disposal are aimed at individual signals, i.e., one mechanism monitors one signal. The mechanisms can monitor

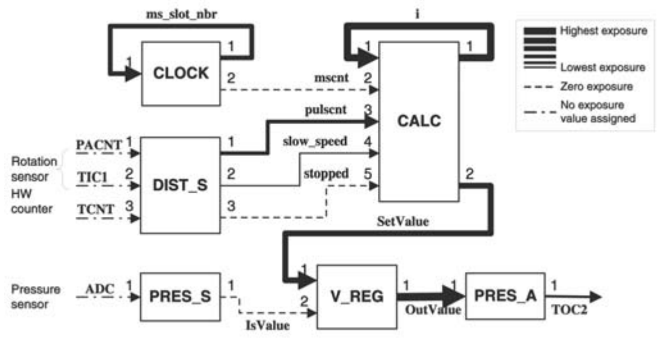


Fig. 15. Propagation analysis: *Exposure* profile.

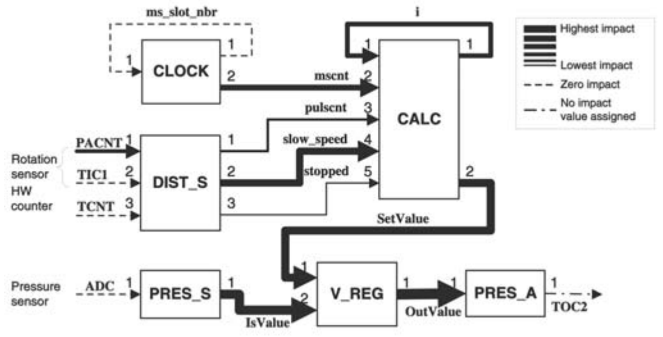


Fig. 16. Effect analysis: *Impact* profile.

continuous signals with regard to minimum and maximum values as well as minimum and maximum rates of change. Discrete signals are monitored with regard to their valid domain and the transition between the values in that domain. The mechanisms have difficulties monitoring Boolean signals as these cannot be modeled very well given the characteristics of the EAs.

As the chosen mechanisms operate at the signal level, we will concentrate on the profiles of the system at signal level. If one were able to replicate entire modules, then the module level profiles may be useful as well.

7.1 Selection 1: Propagation-Based Selection of Locations

We start out by basing our selection on the propagation profile of the system. In the analysis of the profiles of the system, we only need to take into account those signals which are along the direct propagation paths from the input signals to the output signal.

As the EAs we have chosen for our system are aimed at individual signals, we take a closer look at individual permeability values and the signal error exposure values (see Tables 1 and 4, respectively) in order to select the signals to equip with EAs.

The experimentally ascertained exposure of *IsValue* is zero, meaning that errors in *ADC* are unlikely to propagate through PRES_S. Thus, although the permeability of errors from *IsValue* to *OutValue* is quite high (0.896), we do not select *IsValue* as a location for an EA.

We do not select *ms_slot_nbr* as errors in this signal do not propagate into *mscnt*. We do not select *TOC2* either, as

TABLE 5
Obtained Detection Coverage for Errors Injected in System Input

Signal	n_{err}	**EA1**	**EA2**	**EA3**	**EA4**	**EA5**	**EA6**	**EA7**	**Total**
Member of P-set		√	-	√	√	-	-	√	
Member of P&E-set		√	√	√	√	√	√	√	
PACNT	1856	0.218	0.105	-	0.975	-	-	0.005	0.975
TIC1	3712	-	-	-	-	-	-	-	-
TCNT	3712	-	-	-	-	-	-	-	-
DIST_S	**9280**	**0.044**	**0.021**	**-**	**0.195**	**-**	**-**	**< 0.001**	**0.195**
ADC	3840	-	-	-	-	-	-	-	-
PRES_S	**3840**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**
All	**13120**	**0.031**	**0.015**	**-**	**0.138**	**-**	**-**	**< 0.001**	**0.138**

this is a hardware register and any errors here would most probably come from the *OutValue* signal. We do not select *mscnt* as this signal has a zero error exposure. We do not select *slow_speed* as this signal has a low error exposure and the mechanisms we have chosen are not particularly geared at detecting errors in Boolean values.

Based on the results obtained here, we select the following signals as locations for EAs: *SetValue*, i, *OutValue*, and *pulscnt*. The first three are selected based on their high signal error exposure values and the last one as this is the signal which is most likely to be affected by errors in system input. We will call this selection the P set.

7.2 Selection 2: Adding the Effect Profile to the Selection Process

In this section, we will, in addition to the profile provided by the propagation analysis, also make use of the profile provided by the error effect analysis of the software system. Thus, we will now consider not only where errors tend to propagate but also what effect errors have (regardless of whether these errors are likely to occur or not). Previously, we had ascertained that signals *SetValue*, i, *pulscnt*, and *OutValue* were to be guarded by EAs 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 EAs as well as these have very high impact values (see Table 4). The mechanisms we have chosen are implemented in such a way 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 the impact values of the signals, we can decide to place EAs 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 as was the case before), we also select that signal. We will call this selection the P&E set.

8 Comparing the Detection Coverage of the Two Location Selections

In this section, we compare error detection coverage provided by the two sets of EAs using two distinct error models: 1) errors are introduced at the system input signals only and 2) errors are introduced in random locations in memory. It is important to remember at this point that the obtained coverage for any set of error detection mechanisms does not depend on the selected locations alone, but also on the characteristics of the mechanisms. Thus, in this comparison, the main interest is the difference in the number of mechanisms and in coverage rather than actual absolute coverage values.

8.1 Comparison 1: Errors in System Input Signals

We start with an error model where errors are introduced only through the system input signals and are transients, i.e., after one calculation round they are likely to be overwritten with correct values. This may happen if the memory and code of the system is protected or shielded in some way such that external disturbances (e.g., radiation) do not affect those parts. Sensor values may still be perturbed, however.

After having added the EAs to the system, we performed a set of injection experiments. In these experiments, we used the same tool (PROPANE) and setup as we used for obtaining the estimates of the individual error permeability values (as described in Section 6.3), i.e., we injected transient single-bit errors in system input signals.

In Table 5, 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 those 1,856 were injected before the arrestment of an aircraft was completed). The various *EAx* columns show the obtained coverage for each individual EA (a dash indicates zero coverage), calculated as $\frac{n_{det}}{n_{err}}$. The *Total* column is the combined coverage considering all EAs.

TABLE 6
Detection Coverage for Errors Injected Periodically in System RAM and Stack

	RAM		Stack		Total	
Measure	P&E	P	P&E	P	P&E	P
c_{tot}	0.128	0.056	0.042	0.018	0.106	0.046

Each row contains the data for errors injected into one signal/module except. The *All* row shows the coverage obtained considering all signals/modules. The rows containing tick-marks indicate which EAs were part of the P set and/or P&E set, respectively (a tick-mark, $\surd$, indicates membership).

In Table 5, 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 indicated that errors in *TIC1*, *TCNT*, and *ADC* propagated with a very low probability (zero probability for *ADC*) into any of the signals 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 (*SetValue*), EA2 (*IsValue*), or EA7 (*OutValue*) were also detected by EA4 (*pulscnt*). Here, we can also see that both sets obtain the same coverage.

It may seem odd that EA2, which guards *IsValue*, has a nonzero 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*.

From this, we can conclude that if errors can only enter a system via its input signals, making a selection of EA locations based on the propagation profile only is sufficient from an error detection point of view. As fewer mechanisms are needed, this will also reduce the resource requirements.

8.2 Comparison 2: Errors in Random Locations in Memory

Here, we use a more severe error model. 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 milliseconds. The same 25 test cases were used, giving us a total of $200 \cdot 25 = 5,000$ runs with injections. An error is said to be detected if it is detected at least once during the arrestment. These experiments were performed on a real setup of the arrestment system (real hardware, real software, simulated environment—not a simulation run on a desktop computer) using the FIC^3-tool (see [3] for details).

The results are summarized in Table 6. The *RAM*-column contains the coverage values for errors injected into the RAM areas of the modules, and the *Stack*-column the coverage values for errors injected into the stack area. The *Total*-column contains the coverage for all errors. The measure c_{tot} is the total coverage of the EA set.

In the columns marked with *P&E* in Table 6, we can see the coverage values obtained for the EAs selected by utilizing both the propagation profile and the effect profile. The coverage values for the system equipped with the EAs selected using only the propagation profile are shown in the columns marked *P*.

The first observation we make when comparing the results for the two sets is that the coverage for the P set of EAs is lower than the coverage obtained using the P&E set of EAs. For errors injected into RAM, the coverage is just over half that obtained using the full set of EAs and for errors in stack the decrease is even greater. This indicates that using only the propagation profile may prove a weakness if errors are introduced not only via the inputs of a system, but also via internal variables and structures.

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. This fact is also highlighted and discussed in conjunction with Figs. 15 and 16.

From these results, we can conclude that if errors are introduced not only via the system input signals, selecting locations for EDMs based on the propagation profile alone may prove insufficient. The effect profile must also be considered. This finding in itself may not be very surprising as the error model used here short-circuits the propagation process and introduces errors directly in locations where errors are unlikely to appear if they were to propagate from system input only. However, using the profiles of the system enables the developer to make a more educated identification of the most critical parts of the software.

Even though the profiles have been generated with an error model that introduces errors only in the inputs of the modules, the combination of the propagation profiles and the effect profile lessens the impact of this. Thus, even though the fault/error model used for profiling the software should be the same as the model which the system should tolerate, the profiles may be useful even in situations where the system under consideration is subjected to an error model which is different from the one used for profiling.

8.3 Discussion on the Comparison

From the comparison made of the two sets of mechanisms, we can see that, in the first comparison, where errors were assumed to be introduced only at system inputs, they both rendered the same detection coverage. However, as the P set requires fewer resources, it is to be preferred over the P&E set. In the second comparison, where errors were assumed to be introduced at random locations in memory, the P&E set renders a higher coverage than the P set and, therefore, it should be preferred. From these results, we can see that the assumed error model has an impact on which profiles one should use when identifying weaknesses and hot spots.

To generalize the results, we can say that the profiles provided by EPIC cover a layered fault model, an onion model, where we have errors in system input signals at the core and then add errors in intermediate signals as a layer around the core. For the core, the propagation profile will be sufficient, whereas, for the outer layer (which combines the core with errors in intermediate signals), the propagation profile is used in combination with the effect profile. Thus, EPIC provides a composite profiling framework which is able to accommodate this onion model.

9 Discussion on Framework Limitations and Caveats

In this paper, we have presented an analysis framework for error propagation and effect analysis. We have also used it in an example study illustrating its ability to indicate weaknesses and hot spots in a software system. However, there are limitations to the framework which we highlight in this section.

One limitation is that, when defining the basic *error permeability* measure, we have only considered direct influence between one input signal and one output signal of a module. It may be (and, in most practical cases, probably is) the case that the probability of an error propagating from a given input signal to a given output signal of a module is not independent of errors on other input signals. Thus, the *error permeability* values may differ when multiple input signals are erroneous as compared to when only one input signal is erroneous.

Related to that, we also do not take into account covert channels through which errors may propagate, i.e., links between modules which are not explicitly known. A fault may occur which creates a link such that an error can be passed from one module to another through an otherwise nonexistent channel. However, it could be argued that this pertains to the assumed fault model instead of the presented analysis framework.

Moreover, the framework currently only considers error propagation and error effect in a "direct" manner, i.e., we only take into account errors that propagate from their original locations to system output directly. This means that we do not consider errors which might, at one point, propagate out of the system, affect the environment (the controlled entity), and then get fed back into the system via the system input signals. It could be argued that, if the controlled environment has an inherent inertia (i.e., minor perturbations on the output signals of the control system do not affect the environment very much), the probability for this is small. Thus, this may not present a serious problem; however, one should bear this possibility in mind.

Also, at this point, we have in our experiments only considered the basic error model *transient bit flips*, i.e., an error was introduced by flipping the value of one bit in one calculation round. Should other error models be used, the results obtained by propagation and effect analysis may need to be reestimated using the basic approach outlined in EPIC. Even though the use of two distinct profiles lessens the impact of varying error models (as discussed in [14]), one should aim at using an error model as close as possible to the assumed error model for the "real" environment in which the system is designed to operate.

Another issue to discuss here is the cost of performing an analysis using EPIC and whether the approach scales with increasing system size. A module with m inputs and n outputs generates $m \cdot n$ error permeability values. Thus, for a large number of modules, the total amount of data required to produce estimates may be large, depending on the tools used and the format in which results are stored. On the other hand, as the approach is modular, i.e., estimates for modules are produced individually, experiments may run in parallel, which reduces the total amount of calendar time required. In the example shown in this paper, we ran experiments for 60 hours in parallel on up to four PCs equipped with Intel Pentium 166 MHz processors.

It should be noted here that, as we have discussed in the comparison (Section 8), we can adopt an onion model for faults, i.e., we have a core model which is extended with layers. The framework described in this paper accommodates for a core model where errors are introduced in the system inputs only and a layer where we add errors in intermediate signals and locations. Addressing the limitations noted here may result in the addition of more layers to this fault model. For instance, the covert channels mentioned above may be added as a layer outside of the errors in intermediate signals and locations and outside of that layer we might add control flow errors. Thus, one may in the future extend the profiling framework such that it is able to also consider these new layers while still maintaining the current abilities for the core layers.

10 Summary and Conclusions

This paper presents the EPIC framework for analysis of the 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 which take in data as signals and produce output, also as signals. Specifically, the main contributions of this paper are:

Software Profiling. The EPIC framework is able to produce distinct profiles of a given modular software system allowing the assessment of the vulnerability of software modules and signals to upcoming data errors. These profiles are 1) an error propagation-based profile and 2) an error effect-based profile. We introduced the basic measure *error permeability* defined on an input/output signal pair basis from which a set of related measures (both at the signal level and the module level) can be calculated. The framework has four basic measures all related to data errors:

1. exposure,
2. permeability,
3. impact, and
4. criticality.

The first two measures relate to the analysis of error propagation, whereas the last two measures relate to the analysis of error effect. Thus, the framework is capable of providing a software designer with two distinct profiles regarding the software system at hand, upon which design decisions regarding error detection and recovery can be based. As the framework assumes a black-box view of the software, its applicability is not limited to software developed in-house, i.e., it can also be used for software which is provided in libraries where only interface specifications are provided (e.g., COTS components).

Identifying Weaknesses and Hot Spots. Using the profiles obtained by using the EPIC framework, we have shown how one may interpret the results using a set of guidelines in order to indicate weaknesses and hot spots in the analyzed software. We have discussed how to identify those parts of the system which may be a target for dependability efforts. These methods can also pinpoint critical signals and paths in a system. Trade offs regarding these guidelines that might have to be considered are also discussed.

Experimental Estimation Method. We have described an experimental method based on fault injection for obtaining estimates for the error permeability values. This method is based on transient bit-flips occurring at single input signals of the various software modules of the software system. Propagation to the output signals of the given module is detected by doing a comparison with "error-free" reference runs (Golden Run Comparisons).

Example Study. We have conducted an experimental assessment on the software of an embedded control system for aircraft arrestment. The results illustrate that using the presented framework generates knowledge on error propagation and error effect, giving insights into software vulnerabilities that are very useful when designing dependable systems. We also show an example of how the obtained profiles may be used in the selection of locations for error detection mechanisms.

Discussion on limitations. We have identified and discussed some limitations and caveats of the EPIC framework. This discussion also suggests directions for future work regarding the EPIC framework.

Concluding this paper, we state that the presented analysis framework, EPIC, provides a means for software profiling which may provide knowledge pertinent to dependability engineering in software systems.

Acknowledgments

This work was mainly performed at Chalmers University of Technology, Sweden; supported in part by the Volvo Research Foundation (FFP-DCN), by NUTEK (1P21-97-4745), by the Saab Endowment, and by US National Science Foundation Career CCR 9896321. The authors also would like to thank the anonymous reviewers for their insightful comments helping us to improve this paper.

References

[1] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault Injection for Dependability Validation: A Methodology and Some Applications," *IEEE Trans. Software Eng.*, vol. 16, no. 2, pp. 166-182, 1990.

[2] R. Chillarege and N.S. Bowen, "Understanding Large System Failures—A Fault Injection Experiment," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-19)*, pp. 356-363, 1989.

[3] J. Christmansson, M. Hiller, and M. Rimén, "An Experimental Comparison of Fault and Error Injection," *Proc. Int'l Symp. Software Reliability Eng. (ISSRE '98)*, pp. 369-378, 1998.

[4] Gy. Csertán, A. Pataricza, and E. Selenyi, "Dependability Analysis in HW-SW Codesign," *Proc. Int'l Computer Performance and Dependability Symp. (IPDS '95)*, pp. 306-315, 1995.

[5] M. Cukier, D. Powell, and J. Arlat, "Coverage Estimation Methods for Stratified Fault-Injection," *IEEE Trans. Computers*, vol. 48, no. 7, pp. 707-723, July 1999.

[6] J.-C. Fabre, F. Salles, M. Rodriguez-Moreno, and J. Arlat, "Assessment of COTS Microkernels by Fault Injection," *Proc. Conf. Dependable Computing for Critical Applications (DCCA-7)*, pp. 25-44, 1999.

[7] H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-13)*, pp. 98-105, 1983.

[8] S.J. Geoghegan and D. Avresky, "Method for Designing and Placing Check Sets Based on Control Flow Analysis of Programs," *Proc. Int'l Symp. Software Reliability Eng. (ISSRE '96)*, pp. 256-265, 1996.

[9] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Trans. Computers*, vol. 30, no. 3, pp. 215-222, Mar. 1981.

[10] J. Güthoff and V. Sieh, "Combining Software-Implemented and Simulation-Based Fault Injection into a Single Fault Injection Method," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-25)*, pp. 196-206, 1995.

[11] M. Hiller, "Executable Assertions for Detecting Data Errors in Embedded Control Systems," *Proc. Int'l Conf. Dependable Systems and Networks (DSN 2000)*, pp. 24-33, 2000.

[12] M. Hiller, A. Jhumka, and N. Suri, "An Approach for Analysing the Propagation of Data Errors in Software," *Proc. Int'l Conf. Dependable Systems and Networks (DSN 2001)*, pp. 161-170, 2001.

[13] M. Hiller, A. Jhumka, and N. Suri, "On the Placement of Software Mechanisms for Detection of Data Errors," *Proc. Int'l Conf. Dependable Systems and Networks (DSN 2002)*, pp. 135-144, 2002.

[14] M. Hiller, A. Jhumka, and N. Suri, "PROPANE: An Environment for Examining the Propagation of Errors in Software," *Proc. Int'l Symp. Software Testing and Analysis (ISSTA '02)*, pp. 81-85, 2002.

[15] R.K. Iyer and D. Tang, "Experimental Analysis of Computer System Dependability," *Fault-Tolerant Computer System Design*, D.K. Pradhan, ed, chapter 5, Prentice Hall, 1996.

[16] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber, and J. Reisinger, "Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture," *Proc. Dependable Computing for Critical Applications (DCCA-5)*, pp. 267-287, 1995.

[17] J.-C. Laprie, "Dependable Computing: Concepts, Limits, Challenges," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-25)*, pp. 42-54, 1995.

[18] N.G. Leveson, S.S. Cha, J.C. Knight, and T.J. Shimeall, "The Use of Self Checks and Voting in Software Error Detection: An Empirical Study," *IEEE Trans. Software Eng.*, vol. 16, no. 4, pp. 432-443, Apr. 1990.

[19] A. Mahmood, D.M. Andrews, and E.J. McCluskey, "Executable Assertions and Flight Software," *Proc. Digital Avionics Systems Conf. (DASC-6)*, pp. 346-351, 1984.

[20] H. Madeira, M. Vieira, and D. Costa, "On The Emulation of Software Faults by Software Fault Injection," *Proc. Int'l Conf. Dependable Systems and Networks (DSN-2000)*, pp. 417-426, 2000.

[21] C.C. Michael and R.C. Jones, "On the Uniformity of Error Propagation in Software," *Proc. Int'l Conf. Computer Assurance (COMPASS '97)*, pp. 68-76, 1997.

[22] L. Morell, B. Murrill, and R. Rand, "Perturbation Analysis of Computer Programs," *Proc. Int'l Conf. Computer Assurance (COMPASS'97)*, pp. 77-87, 1997.

[23] D. Powell, E. Martins, J. Arlat, and Y. Crouzet, "Estimators for Fault Tolerance Coverage Evaluation," *IEEE Trans. Computer*, vol. 44, no. 2, pp. 261-274, Feb. 1995.

[24] C. Rabéjac, J.P. Blanquart, and J.P. Queille, "Executable Assertions and Timed Traces for On-Line Software Error Detection," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-26)*, pp. 138-147, 1996.

[25] M. Rimén, J. Ohlsson, and J. Torin, "On Microprocessor Error Behavior Modeling," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-24)*, pp. 76-85, 1994.

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

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

[28] F. Salles, M.R. Moreno, J.C. Fabre, and J. Arlat, "MetaKernels and Fault Containment Wrappers," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-29),* pp. 22-29, 1999.

[29] K.G. Shin and T.-H. Lin, "Modeling and Measurement of Error Propagation in a Multimodule Computing System," *IEEE Trans. Computers,* vol. 37, no. 9, pp. 1053-1066, Sept. 1988.

[30] D.T. Smith, B.W. Johnson, and J.A. Profeta III, "System Dependability Evaluation via a Fault List Generation Algorithm," *IEEE Trans. Computers,* vol. 45, no. 8, pp. 974-979, Aug. 1996.

[31] A. Steininger and C. Scherrer, "On Finding an Optimal Combination of Error Detection Mechanisms Based on Results of Fault Injection Experiments," *Proc. Int'l Symp. Fault-Tolerant Computing (FTCS-27),* pp. 238-247, 1997.

[32] 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. 1986.

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

[34] J. Voas, "PIE: A Dynamic Failure-Based Technique," *IEEE Trans. Software Eng.,* vol. 18, no. 8, pp. 717-727, Aug. 1992.

[35] J. Voas, F. Charron, and L. Beltracchi, "Error Propagation Analysis Studies in a Nuclear Research Code," *Proc. Aerospace Conf.,* vol. 4, pp. 115-121, 1998.

Martin Hiller received the MS degree in computer science and engineering and the PhD degree in computer engineering from Chalmers University of Technology, Göteborg, Sweden, in 1996 and 2002, respectively. He is currently with the Volvo Technology Corporation, Gothenburg, Sweden, and is also a part time postdoctoral researcher with the Department of Computer Science at TU Darmstadt, Germany. His research interests include design and assessment of dependable software, fault injection, error detection, and recovery mechanisms, mainly aimed at embedded systems. He received the 2001 William C. Carter Award at DSN for his work on error propagation analysis, and has been coauthor on other award winning publications. He is a member of the IEEE and the IEEE Computer Society.

Arshad Jhumka received the BA and MA degrees in computer science from the University of Cambridge, England. He is currently a PhD candidate at the DEEDS group in the Department of Computer Science at TU Darmstadt, Germany. His research interests and publications have spanned formal verification, robust software design, and validation issues. He is a coauthor of the DSN 2001 Conference Carter Award paper and also received the "Young Researcher" award for his paper at the 2002 International Conference on High Assurance Systems Engineering (HASE). He is a student member of the IEEE.

Neeraj Suri received the PhD degree from the University of Massachusetts at Amherst. He currently holds the TU Darmstadt Chair Professorship in "Dependable Embedded Systems and Software" at TU Darmstadt, Germany, and is also affiliated with the University of Texas at Austin. His earlier academic appointments include the Saab Professorship at Chalmers University of Techniology, Sweden, and earlier at Boston University. His research interests focus on design, analysis, and assessment of dependable embedded systems and software. His current research is emphasizing 1) robustness hardening of software and 2) verification along with experimental validation of protocols, embedded software, and operating systems. His group's research activities have garnered support from US DARPA, US National Science Foundation, ONR, European Commission, NASA, Boeing, Microsoft, Intel, Saab, Volvo, and Daimler Chrysler among others. He is also a recipient of the US NSF CAREER award. He serves as an editor for *ACM Computing Surveys* covering embedded systems and real-time and has been an editor for the *IEEE Transactions on Parallel and Distributed Systems*. He is a member of IFIP WG 10.4 on Dependability, a senior member of the IEEE, and also on the board for Microsoft's Trustworthy Computing Academic Advisory Board. More research and professional details are available at http://www.deeds.informatik.tu-darmstadt.de/.