An Approach for Analysing the Propagation of Data Errors in Software [*]

Martin Hiller, Arshad Jhumka, Neeraj Suri
Department of Computer Engineering
Chalmers University, Göteborg, Sweden
{hiller, arshad, suri}@ce.chalmers.se

Abstract

We present a novel approach for analysing the propagation of data errors in software. The concept of error permeability *is introduced as a basic measure upon which we define a set of related measures. These measures guide us in the process of analysing the vulnerability of software to find the modules that are most likely exposed to propagating errors. Based on the analysis performed with error permeability and its related measures, we describe how to select suitable locations for error detection mechanisms (EDM's) and error recovery mechanisms (ERM's). A method for experimental estimation of error permeability, based on fault injection, is described and the software of a real embedded control system analysed to show the type of results obtainable by the analysis framework. The results show that the developed framework is very useful for analysing error propagation and software vulnerability, and for deciding where to place EDM's and ERM's.*

1. Introduction

As software based functionality becomes pervasive in embedded control systems, software usually comprises numerous discrete modules interacting with each other in order to provide a specific task or service. With an error (as defined in [10]) present in a software module, there is a likelihood that this error can propagate to other modules with which it interacts. Knowing where errors propagate in a system is of particular importance for a number of development activities. Propagation analysis may be used to find the most vulnerable modules in a system, and to ascertain how different modules affect each other in the presence of errors. Furthermore, error propagation analysis also gives an insight on locations in the system that would be best suited for placement of error detection mechanisms (EDM's) and associated error recovery mechanisms (ERM's).

[*]Research supported in part by Volvo Research Foundation (FFP-DCN), by NUTEK (1P21-97-4745), by Saab Endowment, and by NSF Career CCR 9896321

Apart from the technical issues that can be addressed using propagation analysis, there are also issues pertaining to project and resource management. Error propagation analysis may be used as a means of obtaining information for use in decisions on where additional resources for dependability development are necessary and to determine where they would be most cost effective. Software is common not only in applications such as aircraft or other high-cost systems, but also in consumer-based cost-sensitive systems, such as cars. These systems often require both development costs and production costs to be kept low. Analysing error propagation can also complement other analysis activities, for instance FMECA (Failure Mode Effect and Criticality Analysis). Consequently, modules and signals found to be vulnerable and/or critical during propagation analysis might be given more attention during design activities. Thus, error propagation analysis, as a means of both system analysis and resource management, may be a very useful design-stage tool in such systems.

In this paper we present an approach for analysing error propagation in software based systems. Our basic intent is software level error propagation, thus we consider distributed software functions resident on either single or distributed hardware nodes. In our approach, we introduce the measure *error permeability* as well as a set of related measures, and subsequently define a methodology for using these measures to obtain information on error propagation and candidate locations for detection and recovery mechanisms. The basic definition of error permeability is the probability of an error in an input signal permeating to one of the output signals (there is one permeability value assigned between each pair of input/output signals).

Paper organisation: We review related work on error propagation analysis in Section 2. In Section 3 we define the system model used in our proposed approach. The definition of error permeability, and a method for analysing error propagation paths, is the subject of Section 4. How the permeability values relate to the locations of EDM's and ERM's is discussed in Section 5. In Section 6 we describe a method for estimating the error permeability of software modules. An example study and experiment is presented in Section 7, and the results are discussed in Section 8. Summary and conclusions are found in Section 9.

2. Related Work

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

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

An investigation in [12] 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. Our findings do not corroborate this assertion of uniform propagation.

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

3. Software System Model

In our studies, we consider modular software, i.e., discrete software functions interacting to deliver the requisite functionality. A module is a generalised black-box having 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.

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, much like for hardware components on a circuit board. 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 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).

4. Propagation Analysis: Conceptual Basis

In our study we aim to chart the propagation of errors, i.e., how errors propagate and their effect on system operations. Our focus here is on data errors – erroneous values in the internal variables and signals of a system.

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 to improved effectiveness of error detection and handling and the consequent cost/performance-ratio of these mechanisms, as the efforts can be concentrated to those areas of the system to where errors tend to propagate. In propagation analysis, the results are useful even with minimal knowledge of the distribution of the occurring errors, i.e., if one does not know which errors are most likely to appear. Having such knowledge would certainly improve the value of the results, but performing the analysis without it still provides qualitative insights on system error susceptibility.

4.1. Error Permeability

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

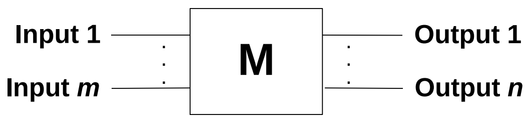

Figure 1. A basic black-box software module with m inputs and n outputs

Consider the software module in Fig. 1. We start with a simple definition of error permeability and refine it successively. For each pair of input and output signals, we can define *error permeability* as the conditional probability of an error occurring on the output given that there is an error on the input. Thus, for input *i* and output *k* of a module **M** we define the *error permeability*, $P^M_{i,k}$, as follows:

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

This measure indicates how *permeable* an input/output pair of a software module is to errors occurring on the inputs. One major advantage of this definition of error permeability is that it is independent of the probability of error occurrence on the input. This reduces the need for having a detailed model of error occurrence. 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 6, we describe an approach for experimentally estimating values for this measure.

Error permeability is the basic measure for characterising error propagation, upon which we develop related refined measures. Accordingly, we define the *relative permeability*, P^M, of a module **M** to be:

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

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.

At this stage, one potential weakness of this measure is that it is not possible to distinguish modules with a large number of input and output signals from those with a small number of input and output signals. This distinction is useful to ascertain as modules with many input and output signals 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 Eq. 2, thereby "punishing" modules with a large number of input and output signals. Thus, we can define the *non-weighted relative permeability*, $\hat{P}^M$, for module **M** as follows:

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

Similar to the relative 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 [17]. 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 Eqs. 2 and 3 are both necessary for analysing the modules of a system. For instance, consider the case where two modules, **G** and **H**, are to be compared. **G** has few inputs and outputs, and **H** has many. Then, if $P^G = P^H$, then $\hat{P}^G < \hat{P}^H$. And vice versa, if $\hat{P}^G = \hat{P}^H$, then $P^G > P^H$.

4.2. Ascertaining Propagation Paths in Inter-Linked Software Modules

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 now considers interactions across modules.

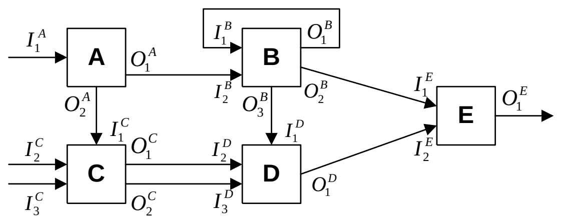

Figure 2. A 5-module example SW system

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 i^{th} input of module **M** is designated I^M_i and the k^{th} output of module **M** is designated O^M_k. External input to the system is received at I^A_1, I^C_2 and I^C_3. The output produced by the system is O^E_1.

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 (each input/output pair of a module has

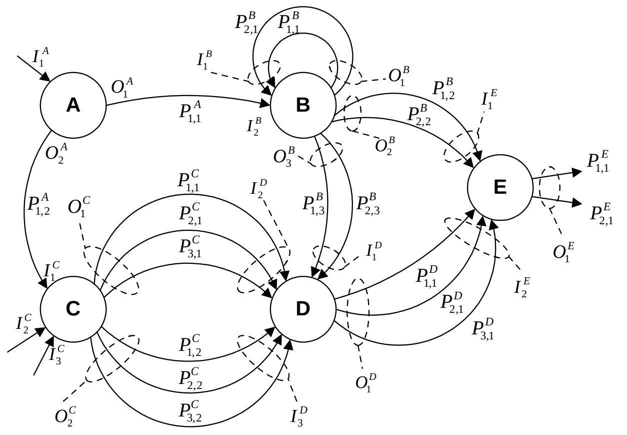

Figure 3. Permeability graph for Fig. 2

an error permeabilit value). Arcs with a zero weight (representing non-permeability from an input to an output) can be omitted. With this permeability graph we can perform two different propagation analyses, namely:

A Backtrack from system o/p to find paths with highest probability of propagation (*Output Error Tracing*), or

B Trace errors from system i/p to find paths these errors will 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:

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

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:

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

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 would use recursive sub-tree generation we would get an infinite number of sub-trees with diminishing probabilities. As all permeability values are ≤ 1, the sub-tree 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. [5, 6, 15] have also utilised similar techniques 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 **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 the probability of an error appearing on I_1^A is $Pr(A1)$, then the P can be adjusted with this factor, giving us $P' = Pr(A1) \cdot P_{1,1}^A \cdot P_{2,2}^B \cdot P_{1,1}^E$. This is the probability of an error appearing on system input I_1^A, propagating through module **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

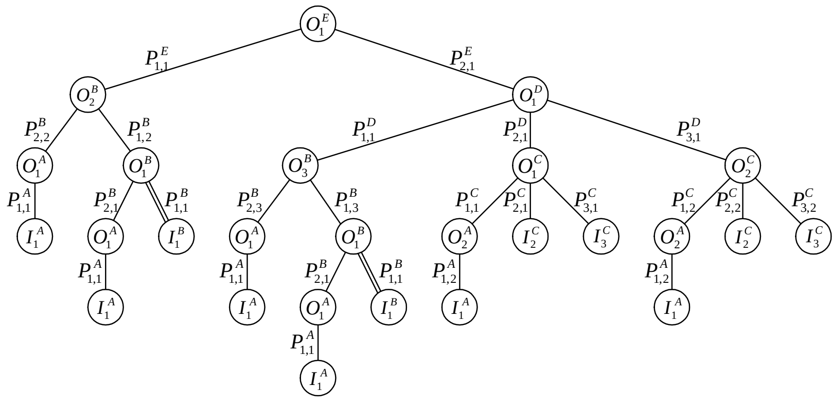

Figure 4. Backtrack tree of system output O_1^E of example system

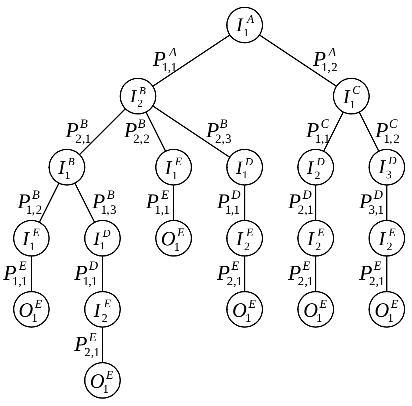

Figure 5. Trace tree for system input I_1^A of example system

backtrack trees, the probability of a path is obtained by multiplying the error permeability values along the path. 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 $P = P_{1,2}^A \cdot P_{1,2}^C \cdot P_{3,1}^D \cdot P_{1,1}^E$.

Again, if we know that $Pr(A1)$ is the probability of an error appearing on I_1^A, then we can adjust P to get $P' = Pr(A1) \cdot P_{1,2}^A \cdot P_{1,2}^C \cdot P_{3,1}^D \cdot P_{1,1}^E$.

5. Relating Error Permeability to Locations for EDM's and ERM's

Using the backtrack and trace trees enables determining two specific aspects: (a) the paths in the system that errors will most likely propagate along to get to certain output signals, and (b) which output signals are most likely affected by errors occurring on the input signals. With this knowledge we can start selecting locations for the EDM's and ERM's that we may want to incorporate into our system based on system reliability/safety requirements.

One problem remains though: once we have the most probable propagation paths, we still have to find the modules along that path that are the best to target with EDM's and ERM's. Earlier, in Eqs. 2 and 3, we had defined two measures, *relative permeability* and *non-weighted relative 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 locations for EDM's and ERM's. 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 *error exposure*, X^M, of a module **M** as:

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

where N is number of incoming arcs and M is the node in the permeability graph, representing software module **M**. This measure does not consider any correlation that may exists between two or more incoming arcs. Since we use this as a relative measure, this is not a concern for us. The *error exposure* is the mean of the weights of all incoming arcs of a node and is bounded as $\frac{1}{N}$. Analogous to the *non-weighted relative permeability*, we can also define the *non-weighted error exposure*, $\hat{X}^M$, of a module **M** as:

$$\hat{X}^M = \sum \text{weight of all incoming arcs of } M \qquad (5)$$

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 (Eqs. 4 and 5) along with the previously defined permeability measures (Eqs. 2 and 3) will be the basis for the analysis performed to obtain information upon which to base a decision about locating EDM's and ERM's. As was the case for the two relative permeability measures, the two exposure measures are used for distinguishing between nodes with a small number of incoming arcs and those with a large number.

The error exposure measure indicates which modules will most probably be the ones exposed to errors propagating through the system. If we want to analyse the system at the signal level and get indications on which signals might be the ones that errors most likely will propagate through, we can define a measure which is the equivalent of the error exposure defined in Eq. 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 B1 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:

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

The interpretation for the signal error exposure is the same as for the error exposure of a module, but at a signal level. That is, the higher a signal error exposure value, the higher the probability of errors in the system being propagated through that signal.

It may be difficult to give strict rules for selecting the EDM and ERM locations. However, some rules of thumb or recommendations can still 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 place EDM's in those modules than in those with lower error exposure. An analogous way of reasoning is valid also for the signal error exposure.

- The higher the error permeability values of a module, the higher the probability of subsequent modules being subjected to propagating errors if errors should pass through the module. Thus, it may be more cost effective to place ERM's in those modules than in those with lower error permeability.

We have now defined a basic analytical framework for ascertaining measures pertaining to error propagation and software vulnerability. Next, we describe how to obtain experimental estimates of the measures and use of our framework on actual software of an embedded control system.

6. Estimating Error Permeability: An Experimental Approach

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 for example [1, 2, 4]. A comprehensive survey of experimental analysis of dependability appears in [9].

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

For this study, we used the Propagation Analysis Environment (PROPANE [8]). This tool enables fault and error injection, using SWIFI (SoftWare Implemented Fault Injection), in software running on a desktop (currently for Windows NT4/2000). The tool is also capable of creating traces of individual variables and different pre-defined 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.

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 **M**, we inject n_{inj} distinct errors in input *i*, and at output *k* observe n_{err} differences compared to the GR's, then we can directly estimate the error permeability $P_{i,k}^M$ to be $\frac{n_{err}}{n_{inj}}$ (see more on experimental estimation in [3] and [14]).

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

The type of injected errors can also effect the estimates. Ideally, one would inject errors from a realistic set, with a realistic distribution. However, as in our framework the measures are mainly used as relative measures, the relevance of the realism provided by the error model is decreased, assuming that the relative order of the modules and signals when analysing permeability is maintained.

7. Experimental Analysis: An Example Embedded System

For an actual application of our proposed methodology on an embedded control system, we have conducted an example study. This study illustrates the results obtained using experimental estimates for error permeability values.

7.1. Target Software System

The target system is a medium sized 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 [19]. The system is illustrated in Fig. 6.

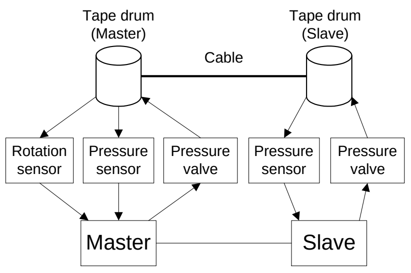

Figure 6. Target used in example study

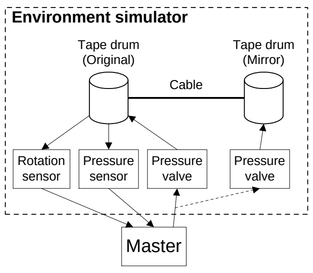

Figure 7. Target and environment simulator

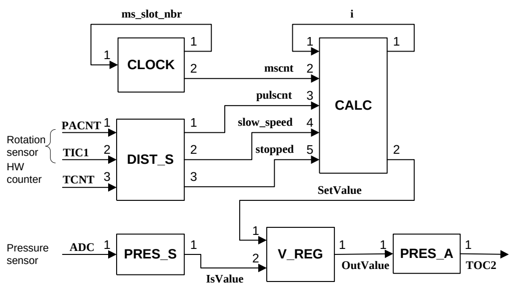

Figure 8. Software structure of target

In our study, we used actual software of the system master and ported it to run on a Windows-based computer. The scheduling is slot-based and non-preemptive. Thus, from the software viewpoint, there is no difference in running on the actual hardware or running on a desktop computer. 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 (as illustrated in Fig. 7).

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. 8. 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. The module specifics are:

CLOCK provides a millisecond-clock, *mscnt*. The system operates in seven 1-ms-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.
CALC uses *mscnt*, *pulscnt*, *slow_speed* and *stopped* to calculate a set point value for the pressure valves, *SetValue*, at six predefined checkpoints along the runway. The checkpoints are detected by comparing the current *pulscnt* with pre-defined *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 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.

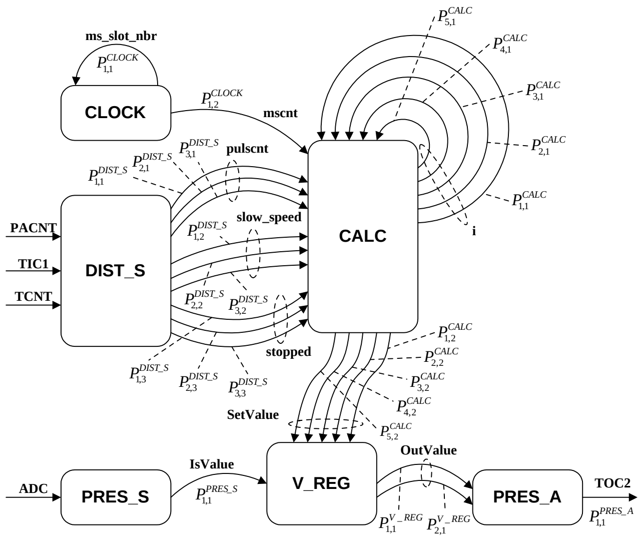

Figure 9. Permeability graph of target

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.

7.2. 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 Sections 4 and 5. The permeability graph is shown in Fig. 9.

In the graph (Fig. 9) 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. 8. 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. 9 we can now generate the backtrack tree for the system output signal *TOC2*, using the

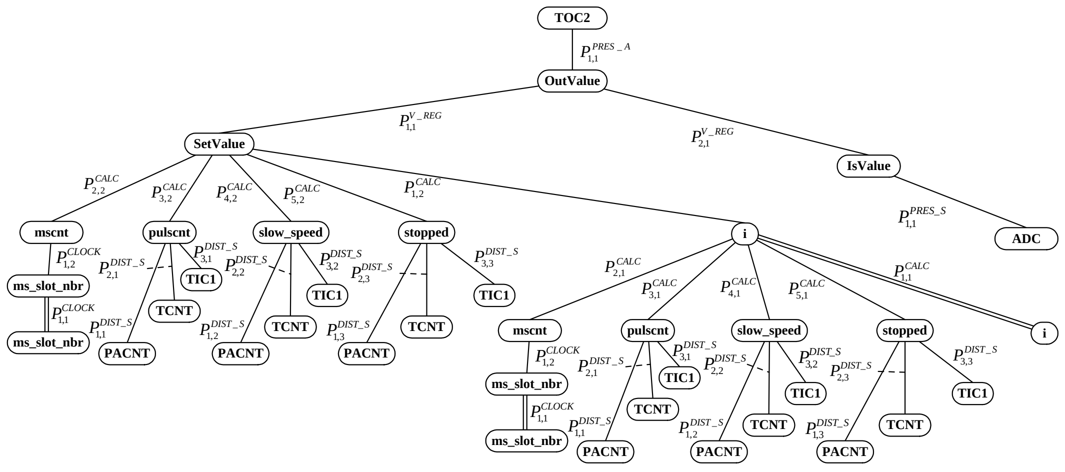
Figure 10. Backtrack tree for output *TOC2*

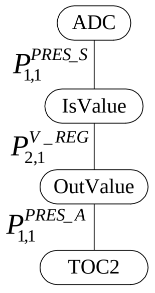
Figure 11. Trace tree for input *ADC*

steps described in Section 4. This tree is shown in Fig. 10.

As illustrated in the backtrack tree (Fig. 10), we have a special relation between the leaves for *ms_slot_nbr* and for *i* and their respective parent. This is because the parent node is also either *ms_slot_nbr* or *i*. Thus, we have an output signal which is connected back to the originating module giving us a recursive relation. In those cases where errors only can enter a system via its main inputs, these branches of the backtrack-trees can be disregarded.

In Figs. 11 and 12, we have the trace trees for system input *ADC* and system input *PACNT*, respectively.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, 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. 12, we do not have a child node from *i* that is *i* itself.

7.3. Experimental Setup

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 non-existent 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 6 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.

The input signals signals were all 16 bits wide. We injected bit-flips in each bit position at 10 different time instances distributed in half-second intervals between 0.5 s and 5.0 s from start of arrestment (although only at one time in each IR). In order to get a realistic load on the system and the modules, we subjected the system to 25 test cases: 5 masses and 5 velocities of the incoming aircraft uniformly distributed between 8,000-20,000 kg, and between 40-80 m/s, respectively. Thus, for each input signal, we conducted $16 \cdot 10 \cdot 25 = 4{,}000$ injections.

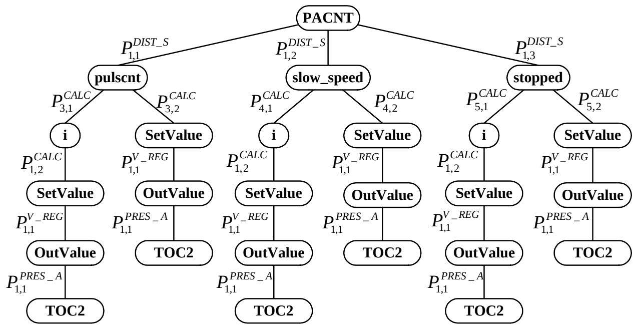
Figure 12. Trace tree for input *PACNT*

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

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

8. Overall Expt. Results and Interpretations

In the target system, we have 25 input/output pairs for which we produced an estimate of the error permeability measure (see Eq. 1) using the method from Section 6. These values (Table 1) form the basis for subsequent results, which are calculated as described in Sections 4 and 5.

Input → Output	**Name**	**Value**
ms_slot_nbr → ms_slot_nbr	$P^{CLOCK}_{1,1}$	1.000
ms_slot_nbr → mscnt	$P^{CLOCK}_{1,2}$	0.000
PACNT → pulscnt	$P^{DIST\_S}_{1,1}$	0.480
TIC1 → pulscnt	$P^{DIST\_S}_{2,1}$	0.004
TCNT → pulscnt	$P^{DIST\_S}_{3,1}$	0.004
PACNT → slow_speed	$P^{DIST\_S}_{1,2}$	0.082
TIC1 → slow_speed	$P^{DIST\_S}_{2,2}$	0.126
TCNT → slow_speed	$P^{DIST\_S}_{3,2}$	0.017
PACNT → stopped	$P^{DIST\_S}_{1,3}$	0.000
TIC1 → stopped	$P^{DIST\_S}_{2,3}$	0.000
TCNT → stopped	$P^{DIST\_S}_{3,3}$	0.000
ADC → IsValue	$P^{PRES\_S}_{1,1}$	0.000
i → i	$P^{CALC}_{1,1}$	1.000
mscnt → i	$P^{CALC}_{2,1}$	0.336
pulscnt → i	$P^{CALC}_{3,1}$	0.791
slow_speed → i	$P^{CALC}_{4,1}$	0.079
stopped → i	$P^{CALC}_{5,1}$	0.209
i → SetValue	$P^{CALC}_{1,2}$	0.457
mscnt → SetValue	$P^{CALC}_{2,2}$	0.666
pulscnt → SetValue	$P^{CALC}_{3,2}$	0.477
slow_speed → SetValue	$P^{CALC}_{4,2}$	0.844
stopped → SetValue	$P^{CALC}_{5,2}$	0.371
SetValue → OutValue	$P^{V\_REG}_{1,1}$	0.884
IsValue → OutValue	$P^{V\_REG}_{2,1}$	0.920
OutValue → TOC2	$P^{PRES\_A}_{1,1}$	0.860

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

Module	P^M	$\hat{P}^M$	X^M	$\hat{X}^M$
CLOCK	0.500	1.000	0.500	1.000
DIST_S	0.079	0.715	-	-
PRES_S	0.000	0.000	-	-
CALC	0.523	5.229	0.313	3.130
V_REG	0.902	1.804	1.408	2.815
PRES_A	0.860	0.860	1.804	1.804

Table 2. Estimated relative permeability and error exposure values of the modules

In Table 2, we obtain weighted and non-weighted relative permeability values (P^M and $\hat{P}^M$, respectively) as well as weighted and non-weighted error exposure values (X^M and $\hat{X}^M$) for each module. The signal error exposure (X^S_s) for each signal is shown in Table 3. The signal error exposure values give us better granularity for deciding which signals we should equip with EDM's or ERM's.

From the backtrack tree in Fig. 10, 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 4 depicts the thirteen paths that acquired weights greater than zero (the paths along which errors might propagate).

Signal	X^S_s
SetValue	2.815
i	2.415
OutValue	1.804
ms_slot_nbr	1.000
TOC2	0.860
pulscnt	0.488
slow_speed	0.225
IsValue	0.000
mscnt	0.000
stopped	0.000

Table 3. Estimated signal error exposures

Path/Product	**Weight**
$P^{CALC}_{1,1} P^{CALC}_{1,2} P^{V\_REG}_{1,1} P^{PRES\_A}_{1,1}$	0.34743
$P^{DIST\_S}_{1,1} P^{CALC}_{3,2} P^{V\_REG}_{1,1} P^{PRES\_A}_{1,1}$	0.17406
$P^{DIST\_S}_{1,1} P^{CALC}_{3,1} P^{CALC}_{1,2} P^{V\_REG}_{1,1} P^{PRES\_A}_{1,1}$	0.13191
$P^{DIST\_S}_{2,2} P^{CALC}_{4,2} P^{V\_REG}_{1,1} P^{PRES\_A}_{1,1}$	0.08085
$P^{DIST\_S}_{1,2} P^{CALC}_{4,2} P^{V\_REG}_{1,1} P^{PRES\_A}_{1,1}$	0.05261
$P^{DIST\_S}_{2,3} P^{CALC}_{4,2} P^{V\_REG}_{1,1} P^{PRES\_A}_{1,1}$	0.01091
$P^{DIST\_S}_{2,2} P^{CALC}_{4,1} P^{CALC}_{1,2} P^{V\_REG}_{1,1} P^{PRES\_A}_{1,1}$	0.00346
$P^{DIST\_S}_{1,2} P^{CALC}_{4,1} P^{CALC}_{1,2} P^{V\_REG}_{1,1} P^{PRES\_A}_{1,1}$	0.00225
$P^{DIST\_S}_{2,1} P^{CALC}_{3,2} P^{CALC}_{1,1} P^{V\_REG}_{1,1} P^{PRES\_A}_{1,1}$	0.00145
$P^{DIST\_S}_{3,1} P^{CALC}_{3,2} P^{V\_REG}_{1,1} P^{PRES\_A}_{1,1}$	0.00145
$P^{DIST\_S}_{2,1} P^{CALC}_{3,1} P^{CALC}_{1,2} P^{V\_REG}_{1,1} P^{PRES\_A}_{1,1}$	0.00110
$P^{DIST\_S}_{3,1} P^{CALC}_{3,1} P^{CALC}_{1,2} P^{V\_REG}_{1,1} P^{PRES\_A}_{1,1}$	0.00110
$P^{DIST\_S}_{3,2} P^{CALC}_{4,1} P^{CALC}_{1,2} P^{V\_REG}_{1,1} P^{PRES\_A}_{1,1}$	0.00047

Table 4. The thirteen non-zero propagation paths and their weights

Comparing the propagation paths from Table 4 with the signal error exposure values from Table 3, we note that the signals with highest exposure values (*SetValue*, *i*, and *OutValue*) are also part of the non-zero propagation paths. From the results obtained by system analysis and the experiments, we can make the following salient observations:

OB1. The modules DIST_S and PRES_S have no error exposure values as they only receive system input signals, i.e., from external sources. This does not mean that these modules will never be exposed to errors on their inputs, but rather that the error exposure is dependent on the probability of errors occurring in the various external data sources. The modules with the highest non-weighted error exposure are the CALC module and the V_REG module. This would indicate that these two modules are central in the system and that they would be good candidates for EDM's and ERM's.
OB2. Note that permeability estimates for errors going from the inputs of DIST_S to its output *stopped* are all zero. It seems as though DIST_S has a built-in resiliency against errors making it non-permeable to errors in that particular output. The reason for this may be that although injected errors can alter the perceived velocity, it is hard to make it zero.
OB3. The permeability of PRES_S (which has only one input/output-pair) is also zero. In previous experiments [7] we investigated the efficiency

of a number of error detection mechanisms based on the concept of executable assertions, e.g., [11, 16]. These results showed that we could devise a detection mechanism that, with a very high probability, detected errors in the signal *IsValue*. Taking into account the knowledge gained by the propagation analysis performed here, we can now say that even though the detection probability for that mechanism was high, it would not be cost effective to incorporate it into the system since the signal it monitors has a very low error exposure. So, although the permeability of errors on the *IsValue* signal to the *OutValue* signal is quite high (0.920), it may be wiser to spend resources on a mechanism that has a higher probability of actually being used, even though it may not detect errors as well. This clearly illustrates that not only are the detection capabilities of EDM's important, the locations are equally important. This is an important sensitivity to determine for each target system. Thus, it should be preferred to put a detection mechanism with a slightly lower detection probability at a location where errors very likely pass by during propagation rather than placing a mechanism with a very high detection probability at a location which seldom is exposed to propagating errors.

OB4. Based on the results obtained here, we would select the following signals as locations for ERM's: *SetValue*, *i*, *OutValue*, and *pulscnt*. The first three are selected since they had the highest signal error exposure and were part of the propagation paths with the highest weights, and the last one as this is the signal which is most likely to be affected by errors in system input. We would not select *ms_slot_nbr* as this signal is independent of all signals, thus, errors will not show up in this signal unless they originate here. We would not select *TOC2* either, as this is a hardware register and any errors here would most probably come from the *OutValue* signal.

OB5. *SetValue* and *OutValue* are part of all propagation paths in Table 4. Therefore they are strong candidates for locations of ERM's, since if errors can be eliminated here, the system output will not be affected (given total success for the recovery mechanisms). Given that CALC has the highest relative permeability it would probably be a very strong candidate for recovery mechanisms in order to avoid incoming errors to propagate through the module to other modules.

OB6. Even though DIST_S has a lower relative permeability than V_REG, one should consider placing recovery mechanisms in DIST_S as they would form a barrier to errors coming in from external data sources, thereby decreasing the probability of errors entering the system in the first place.

9. Summary and Conclusions

Overall, we have presented a generalised framework for analysing the propagation of data errors in software systems and the vulnerability of software modules. The framework introduces the basic measure *error permeability* from which a set of related measures can be calculated. The measures calculated from the error permeability allow for an assessment of the vulnerability of software and its modules.

The framework also contains methods for obtaining propagation paths and ordering them according to their likelihood. Furthermore, the measures and analysis results are shown to be useful input to the process of selecting locations in the software would be suitable for error detection mechanisms and error recovery mechanisms. These methods can also pinpoint critical signals and paths in a system.

We conducted an FI driven experimental assessment on the software of a medium-sized embedded control system. The results clearly show that using the presented framework generates knowledge on error propagation and software vulnerability that is very useful when designing dependable systems. The results also illustrate that by incorporating less efficient detection and recovery mechanisms in locations which have high error exposure instead of very efficient mechanisms which are seldom exposed to errors one would most likely get a better cost-performance ratio.

Future work includes analysing the effect of workload as well as error models on the permeability estimates for varied embedded software based systems to further investigate the applicability and scope of the framework.

References

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

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

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

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

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

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

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

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

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

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

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

[12] Michael C. C., Jones R. C., "On the Uniformity of Error Propagation in Software", *Proc. COMPASS'97*, pp. 68-76, 1997.

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

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

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

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

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

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

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

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

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

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