Spectrum-based Health Monitoring for Self-Adaptive Systems

Éric Piel, Alberto Gonzalez-Sanchez, Hans-Gerhard Gross and Arjan J.C. van Gemund
Department of Software Technology
Delft University of Technology
The Netherlands
{e.a.b.piel, a.gonzalezsanchez, h.g.gross, a.j.c.vangemund}@tudelft.nl

***Abstract*—An essential requirement for the operation of self-adaptive systems is information about their internal health state, i.e., the extent to which the constituent software and hardware components are still operating reliably. Accurate health information enables systems to recover automatically from (intermittent) failures in their components through selective restarting, or self-reconfiguration.**

This paper explores and assesses the utility of Spectrum-based Fault Localisation (SFL) combined with automatic health monitoring for self-adaptive systems. Their applicability is evaluated through simulation of online diagnosis scenarios, and through implementation in an adaptive surveillance system inspired by our industrial partner. The results of the studies performed confirm that the combination of SFL with online monitoring can successfully provide health information and locate problematic components, so that adequate self-* techniques can be deployed.

I. Introduction

It is generally accepted that all but the most trivial systems will inevitably contain residual defects. Adaptive and self-managing systems acknowledge this fact through deployment of fault tolerance mechanisms that are able to react adequately to problems observed during operation time. A fundamental quality of such a system is, therefore, its ability to constantly maintain internal health information of its constituent parts, and isolate the root cause of a failure in case the system health decreases. Once the fault is isolated, and the problematic component(s) identified, the system can unleash its full range of inbuilt self-protection, -adaptation, -reconfiguration, -optimization, and -recovering strategies to resume its normal operation.

A system that is able to reason about its own health and pinpoint problematic components requires built-in monitoring techniques, which enable the observation of deviations from its nominal behaviour, and built-in fault localisation strategies, that permit the system to convict or exonerate a potentially faulty component. Although, up to now, SFL has only been applied offline, it can be used online, i.e., in combination with specifically designed monitoring approaches. To the best of our knowledge, SFL is the most light-weight fault localization technique available to be used for the provision of health information and for identifying problematic components in adaptive systems.

In this paper, we make the following four contributions. (1) We demonstrate how SFL can be applied to online fault diagnosis. (2) We present two specific *observation approaches* that support efficient and effective online diagnosis through time-/transactional separation. (3) We develop and assess a simple but effective *sliding window technique* that helps to keep the diagnosis in sync with the currently observed health state of the system. (4) We assess our proposed techniques in simulations as well as in a real industrial case study.

Section II outlines SFL and the monitoring approach on which it relies for performing online diagnosis. Section III describes how we performed the simulations for evaluation. Section IV depicts the observation and windowing techniques used for the continuous health information. Section V presents the case study. Section VI discusses related work, and section VII summarizes and concludes the article.

II. Fault Diagnosis

The objective of fault diagnosis is to pinpoint the precise locations of faults in a system by observing the system's behaviour. Before delving into the usage of the SFL approach for online fault localisation, and the provision of health information, let us introduce SFL in its offline version. Typical *active testing* cannot be applied online, because of interference, so that continuous validation must come from observations provided by monitors. This may also be referred to as *passive testing*. The following inputs are usually involved in SFL approaches:

- A finite set $\mathcal{C} = \{c_1, c_2, \ldots, c_j, \ldots, c_M\}$ of M "components" (e.g., source code statements, functions, classes) which are potentially faulty. We will denote the number of faults in the system as M_f.
- A finite set $\mathcal{T} = \{t_1, t_2, \ldots, t_i, \ldots, t_N\}$ of N tests (observations in the online version) with binary outcomes $O = (o_1, o_2, \ldots, o_i, \ldots, o_N)$, where $o_i = 1$ if test t_i failed, and $o_i = 0$ otherwise.
- A $N \times M$ *activity matrix*, $A = [a_{ij}]$, where $a_{ij} = 1$ if test t_i involves (covers) component c_j, and 0 otherwise. Each row a_i of the matrix is called a *spectrum*. Due to the continuous nature of the target systems in online health monitoring, an important consideration is how to manage the coverage matrix A, which is discussed in Sect. IV.

The output of fault localisation, is a *diagnosis*, which is a ranking of the components ordered according to their assumed

health state within the system. This ranking is an indicator for the likelihood of the components containing the fault(s).

In program debugging, the granularity of a *component* is often very small, typically at the statement level, since SFL benefits from variations in program control flow. However, in an online context, we selected a larger grain size as components, i.e., source code function (or source code procedure). This still permits to monitor a system and to take the appropriate actions in case of degradation, while it reduces the performance overhead, and represents a more realistic component granularity for large systems[1].

An important property of any diagnosis approach is its diagnostic performance, representing how well the diagnosis algorithm can pinpoint the true root cause of an observed problem. In SFL, this is expressed in terms of a metric C_d that measures the theoretical *effort* still needed for a diagnostician to find all faulty components after reading the generated diagnosis [5]. In an autonomic context this metric describes the (un)certainty of a diagnosis when making decisions such as aborting a mission, changing a component, etc. C_d measures *wasted effort*, independent of the number of faults M_f in the system, to enable an unbiased evaluation of the effect of M_f on C_d. Thus, regardless of M_f, $C_d = 0$ represents an ideal diagnosis technique (all M_f faulty components are ranked at the top, and no effort is wasted for a human to check healthy components), while $C_d = M - M_f$ represents the worst diagnosis technique (checking all $M - M_f$ healthy components before the M_f faulty ones). For example, consider a diagnosis algorithm that returned the ranking $\langle c_{12}, c_5, c_6, \ldots \rangle$, while c_6 contains the actual fault. This diagnosis leads the developer to inspecting c_{12} and c_5 first. As both components are healthy, C_d is increased by 2, and the next component to be inspected is c_6. As it is faulty, no more effort is wasted and $C_d = 2$. To ease comparison between systems, a *relative wasted effort* is often used: $\frac{C_d}{M - M_f}$.

A. Statistical Fault Diagnosis

Statistical SFL is a well-known approach originating in software engineering [4], [15]. Here, fault likelihood l_j (and thus assumed health) is quantified in terms of *similarity coefficients* (SC). SC measure the statistical similarity between component c_j's test coverage $(a_{1j}, \ldots, a_{Nj})$ and the observed test outcomes, $(o_1, \ldots, o_N)$. It is computed by four values $n_{pq}(j)$ counting the number of times a_{ij} and o_i form the combinations $(0,0), \ldots, (1,1)$, respectively, i.e.,

$$n_{pq}(j) = |\{i : a_{ij} = p \wedge o_i = q\}| \;\; p, q \in \{0, 1\} \qquad (1)$$

For instance, $n_{10}(j)$ and $n_{11}(j)$ are the number of tests in which component c_j is executed, and which passed or failed, respectively. The four counters sum up to the number of tests N. Two commonly known SCs are the Tarantula [15], and Ochiai [4] similarity coefficients, given by

$$\text{Tarantula:} \qquad l_j = \frac{\frac{n_{11(j)}}{n_{11(j)}+n_{01(j)}}}{\frac{n_{11(j)}}{n_{11(j)}+n_{01(j)}} + \frac{n_{10(j)}}{n_{10(j)}+n_{00(j)}}} \qquad (2)$$

$$\text{Ochiai:} \qquad l_j = \frac{n_{11(j)}}{\sqrt{(n_{11(j)}+n_{01(j)}) \cdot (n_{11(j)}+n_{10(j)})}} \qquad (3)$$

Ordering the components by their l_j, results in the ranking of the diagnosis algorithm.

Despite their lower diagnostic accuracy [5], SC are ideal for online diagnosis due to their ultra-low computational complexity (compared with probabilistic diagnosis approaches based on Bayesian reasoning). Another advantage is the fact that SC are incremental, so there is no need to compile a (possibly huge) test coverage matrix. Only the counters n_{pq} must be kept per component. Finally, unlike Bayesian approaches, statistical SFL is robust w.r.t. uncertainties in the test outcomes. While all techniques tolerate false negatives (i.e., a test involving a faulty component and not returning a failure), statistical approaches are more robust w.r.t. false positives, which is essential in online monitoring as the oracles are often less sophisticated than in offline testing.

B. Monitoring

The main difference of this work compared to previous application of SFL is the use of *online monitoring* instead of *offline testing* for the provision of observations and hence health information. A monitor is a specific component in the system that observes and assesses the correctness of the business logic without interfering through active test inputs. Monitors are executed along with the business logic, merely adding performance overhead. Monitoring is well understood, easy to apply, and event-based, due to its passive nature, e.g., triggered by the arrival of a new data, or a timer interrupt. A monitor observes data or behaviour in specific predefined locations and decides based on built-in oracle logic whether an observation is expected (*pass*) or unexpected (*fail*), for example through checking invariants, or through comparison with a state model.

III. Online SFL Simulation

For initial illustration and evaluation of online SFL we use synthetic system simulations next to an actual case study. Simulations can be executed quickly (e.g., for our case study system we can simulate one hour of operation in just a few seconds). They avoid implementation details which could cause noise in the observations (e.g., monitors with false positives), and they allow to vary many properties of a base system, in order to generalise the findings according to many different (synthetic) system configurations. The simulations use models of the system under consideration in terms of different topologies of the surveillance system used as case study. The different system topologies generate outputs similar to the ones used by the actual SFL diagnosis algorithm, i.e., a ranking of the components according to their assumed health

[1] In reality the granularity would reflect the level at which components can be plugged in and out the system dynamically.

for a complete period of execution of the simulation. Simulator and example models are available for download[2].

A. System Modeling for Simulation

We use two layers of representation for simulation: a topological layer and an execution layer. The topological layer models the system in terms of the relation between each business logic component, the location of the monitors, and the location of the faults. Each component has a health variable $0 \leq h \leq 1$ indicating its likelihood to generate the output expected from the specification. By default the value is 1, meaning it is healthy. A fault in a component is simply inserted by setting h to a low value, representing the likelihood the fault does *not* cause a failure when the component is executed. The topology is represented by a Component Interaction Graph (CIG) [28] with components as vertices and calls as edges. Monitors are like normal components in the system.

Figure 1 shows an example CIG, with 7 business logic components and 3 monitors (A, B, and C). Component 2 is set to be faulty, with $h = 0.4$ ($h = 1$ for the other components). This CIG can be read as control-flow graph. The model represents a data-flow system where component 1 receives the inputs and passes them on to the other components.

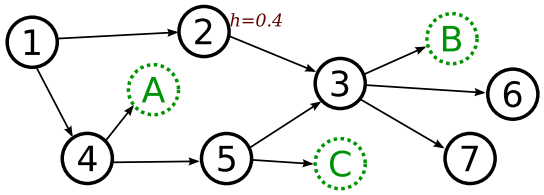

Fig. 1. Example topological layer with 7 business logic components and 3 monitors.

The behaviour in the topology is defined by the execution layer. It contains a set of *execution paths*, each comprising a list of components in the order they are executed. A path must be consistent with the topology of the system: components may be executed in a sequence if the topological model defines this through edges. Fig. 2 shows an example with three paths through the model (from Fig. 1).

A *goodness* attribute g is added to every monitor. This attribute represents the likelihood that a monitor's outcome is *pass* and the monitored sub-path is not leading to failure, even if there was a fault. This is based on the *Propagation, Infection, Execution* notion by Voas [24], representing the fact that a failure in a component can still lead to a correct output in the subsequent component. It makes the simulations more realistic (and more difficult for the SFL algorithm by introducing more false negatives). g should be set to $h < g < 1$ for the lowest h in the considered path. The same monitor can have different values of g in different execution paths. Finally, a frequency is associated with each execution path, representing how often this path is taken during the execution of the system (Fig. 2).

[2]http://swerl.tudelft.nl/bin/view/Main/SOFL

$$
\begin{array}{ll}
1 \rightarrow 2 \rightarrow 3 \rightarrow B_{g=0.9} \rightarrow 3 \rightarrow 6 \rightarrow 3 \rightarrow 7 & f = 0.2Hz \\
1 \rightarrow 4 \rightarrow A_{g=1} \rightarrow 4 \rightarrow 5 \rightarrow C_{g=1} \rightarrow 5 \rightarrow 3 \rightarrow B_{g=1} & f = 1.2Hz \\
1 \rightarrow 2 \rightarrow 3 \rightarrow B_{g=0.6} \rightarrow 3 \rightarrow 7 & f = 3Hz
\end{array}
$$

Fig. 2. Example of the execution layer of a model of a system with 3 execution paths.

All 3 paths (Fig. 2) start with component 1 as entry point to the system. The second path never covers the faulty component, i.e., all monitors along this path have $g = 1$, always reporting a pass (2). The third path is executed on average 15 times more often than the first one. In addition, false positive and false negative monitor outcomes can be simulated with this setup, with $h < 1$ (indicating the probability that the outcome is inverted).

B. Simulated Behaviour

The simulation of a path consists in traversing all components of the path, marking them as covered (adding the component to the spectrum, see Section IV-A). If a monitor is activated, current time, coverage (spectrum), and outcome are logged. The outcome is randomly generated from the g likelihood associated with the monitor. The simulation ends after a preset period of time, and the monitor logs are provided as simulation output.

Figure 3 shows example logs generated from a simulation of the system shown in Figure 2 over a period of one second. Each line corresponds to the log of one monitor triggered in the execution path: a spectrum and an outcome (simulation started at time 0). The monitor logs are passed to the fault localisation algorithm for computing the estimated fault location (i.e., the assumed health of the components). Comparing the output of the fault localisation with the actual position of the fault in the model yields the diagnostic performance C_d (described in Section II).

Time	1	2	3	4	5	6	7	Outcome
0.0	1	1	1					Fail
0.0	1			1				Pass
0.0	1			1	1			Pass
0.0	1		1	1	1			Pass
0.0	1	1	1					Pass
0.33	1	1	1					Fail
0.66	1	1	1					Pass
0.83	1			1				Pass
0.83	1			1	1			Pass
0.83	1		1	1	1			Pass
1.0	1	1	1					Fail

Fig. 3. Example of spectra and monitor outcomes generated from a simulation of 1 second.

C. Generation of System Configurations

As basis for creating many simulations, we used the topology of the surveillance system from the case study in Section V. It comprises 63 components for the business logic, scoped into six main modules (Fig. 10). For each of the 63 components, a configuration was generated with that component being the faulty one with a $h = 0.1$ (where

specified, a set was also created with $h = 0.9$). For each fault location, 10 different system configurations were generated by randomly placing 15 monitors, and producing a set of 20 execution paths (with random frequencies between 0.2 Hz and 50 Hz). Therefore, in the following section, each result presented corresponds to the average result of 630 system configurations.

IV. Continuous Fault Localisation

Applying SFL online brings up two issues: (1) the range of a spectrum (discussed in Subsect. IV-A), and (2) the adequacy of the diagnosis with the current system behaviour (discussed in Subsect. IV-B). In the offline version tests are independent, so that start and end of an interaction and its component coverage are clear, as well as associated inputs and outputs. However, in the case of continuous diagnosis these boundaries disappear, or, at least, become blurred. In this section, we present the challenges in adapting SFL to be used in an online context, and propose solutions to overcome them.

A. Spectrum sampling

In many cases, interactions in a live system are not clearly separable by time or space boundaries (such as a complete test transaction in testing). Input stimuli are continuously arriving and the system responds accordingly changing its internal state and/or producing some output. For example, in our case study, input messages arrive at any time, and sometimes simultaneously in separate threads. Previous inputs influence the behaviour of a component either explicitly such as in a database, or implicitly by affecting its internal state. Every time a system monitor is covered in an execution path, it generates a new spectrum, i.e., coverage of components as row in the coverage matrix A, and its outcome (pass/fail) is added to the vector O. Component coverage is recorded since the system was started, and after a short period of operation, the coverage matrix will contain only 1's: "everything covered". Although this approach would guarantee a strong causal relationship between fault execution and failure observation (i.e., if a failure is observed, the spectrum will contain the fault information), a solid 1's spectrum does not provide any diagnostic information for the SFL, because it infers the diagnosis from differences of the various spectra in the coverage matrix A and the outcome O.

The curves termed *time inf* of Figures 4 and 5 show the effect of the spectrum observation as used for assessing the diagnostic cost throughout the execution of the system. The cost is almost constant for the entire time at 0.5, meaning the correct fault location is on average in the middle of the provided ranking. Guessing the fault locations randomly would yield a similar performance.

The coverage of components represented as binary values in the spectrum must be reset regularly, in order to provide a meaningful diagnosis. In the following we discuss some possible alternatives for resetting or limiting the spectrum in terms of time and space, in order to impose some artificial transaction boundaries on the system execution.

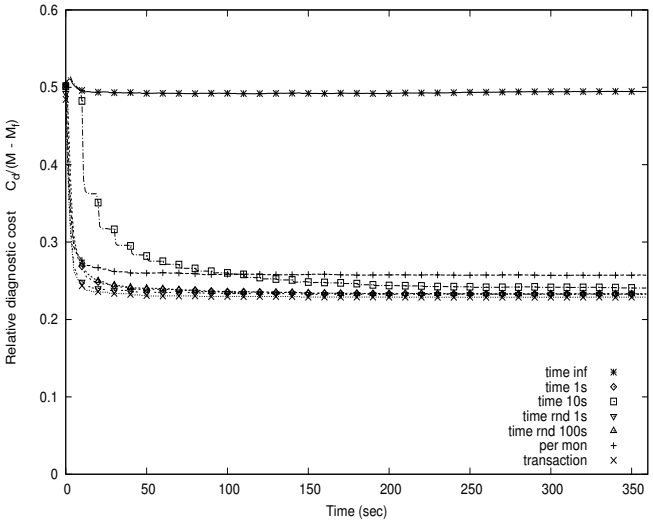

Fig. 4. Average diagnostic cost along the time of observation for various observation policies, with one fault expressed as $h = 0.1$.

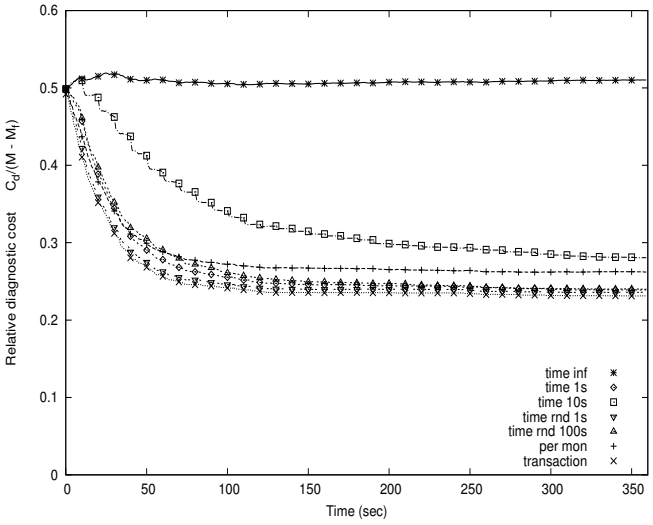

Fig. 5. Average diagnostic cost along the time of observation for various observation policies, with one fault expressed as $h = 0.9$.

1) Data-oriented Spectra: The original idea of SFL and offline-testing is that each outcome of a test is associated with a list of covered components. A fail outcome means that the list of covered components are more suspect to contain the fault, and a pass outcome means that the covered components are less likely suspects.

The *data-oriented spectra* technique associates with each data object in the system additional meta-data that contains information about which components were involved in the processing of that data object. This approach guarantees enforcement of the causal relationship between fault involvement and failure, albeit to the cost of a large storage overhead. A clear disadvantage is the necessity of modifying the system implementation for performing the spectra updates whenever data is created or modified (including all internal state information).

Moreover, for each spectrum update location, an analysis must be performed in order to determine the origin of every new data item. It would be possible to automate the analysis as it is a problem related to definition-use data dependencies [11]. However, it goes directly against the philosophy of applying SFL, which is to avoid white box knowledge of a component altogether.

Despite being the most advanced approach, given the complexity of analysis required and the technology limitation, we only implemented it on a part of the case study system presented in Section V. In order to overcome these major practical difficulties, we investigate other techniques that are easier to realise.

2) Separate-per-Monitor: A monitor validates the correctness of a specific component transaction in the system, corresponding to particular interactive functionality. The provision of an outcome through the monitor correlates to the end of this transaction. The beginning of the next transaction observed by the monitor, therefore, can be approximated to the time just after the outcome is generated. This policy assigns a separate spectrum to every monitor. Each time a component is involved in the execution of a transaction, the current spectrum of every monitor is updated. When a monitor generates an outcome, its associated spectrum is used as a row for the matrix A and is then completely reset to zero.

Figures 4 and 5 show, based on simulation, the effectiveness of this policy (graphs termed *per mon*). Although, it does provide some improvements over a spectrum which is never reset (the diagnostic cost converges towards 0.35 instead of 0.5), we will see that other policies can provide a lower diagnostic cost.

Separate-per-monitor has two drawbacks: (1) Loose relationship between fault and failure. For example, due to internal state, one execution of a faulty component could cause a failure multiple times, but only the detection of the first failure will produce a spectrum that encompasses the faulty location. (2) An execution that happens after the observation is considered part of the next observation, i.e., components that are triggered by the interaction's output are also in the spectrum, breaking the causality expectation, which lowers the diagnostic information that could be extracted by SFL.

The next policy strives to solve this second drawback.

3) Transactional: This policy is based on the *separate-per-monitor* policy but uses additional information to further limit the scope of the spectrum of each monitor to the components involved in the monitored interaction[3]. Only the bits of the components that are explicitly part of the interaction are considered. The list of the components in the interaction linked to each monitor is provided before the execution of the system (and updated after each modification). It is either manually created by the user (the developer of the monitor, most likely), or it could be determined by code or configuration analysis.

Figures 4 and 5 depict this policy labelled *transaction*. It is the most effective policy with diagnostic cost converging towards 0.31. However, if a fault modifies how components interact (i.e., the control flow is modified), the difference between the model and the implementation could lead this policy to omit the faulty component from every spectrum associated with a *fail* outcome. In this special case, the quality of the diagnosis would be adversely affected. In order to avoid relying on pre-analysis of the system we investigate a technique requiring less information about the system.

4) Time Frame: The *time frame* policy uses expiration of time as transaction boundary to establish causality between components covered and monitor outcome. Over a given time period, the component activity is recorded into a global "current spectrum". When the time expires, the bits of the involved components are reset and the recording of a new current spectrum is started. Every monitor outcome during this period, is associated with the current spectrum.

Time frame-based sampling avoids spectra with too many 1's if the time window is properly adjusted to the working speed of the system. However, it does not enforce strong fault-failure relationship. If an error stays dormant during one time frame, and only manifests as failure in a subsequent one, the faulty component may not be in the spectrum, and, hence, the fault-failure relation is not established. This can be mitigated by using larger time frames, although then, we run the risk of producing spectra with too many covered components, thus, loosing diagnostic information.

Figures 4 and 5 show the effect of this policy with time frame periods of 1 and 10 s. All periods converge to the same diagnostic cost, with the shorter ones converging faster[4].

The trade-off between frame length and diagnostic information must be considered. A sensible approach is to use a random frame length. After expiration of time frame, the length of the next frame is determined randomly within reasonable bounds. In our experiment an exponential distribution (with a mean of 1 s) is used in order to compensate for long time frames that would take much more total time in a uniform distribution. Figure 5 illustrates that this technique yields as good results as the best fixed time frame method, and almost as good results as the *transactional* method, which both require additional knowledge about the system. Simulations with different average periods (0.1 to 100 s) provide almost identical diagnostic costs. While the average period must still be selected according to the system under observation, it can be relatively roughly estimated.

Looking at the pro's and con's, we recommend that the observation policy should be selected according to the system context: if it is possible to gather precise information on which interaction is observed by a monitor, then the transactional policy should be applied. Otherwise the randomized time frame policy should be implemented, with just enough validation to ensure the average period is adapted to the system.

[3]Each execution of the interaction can be considered a *transaction*, hence the name of the policy.

[4]The model does not allow to simulate very short periods, which could lead to resetting the spectrum while an execution path is not finished yet, because each execution path is simulated as executing instantaneously.

B. Spectrum Matrix Size

In offline-SFL, the size of the spectrum matrix and the error vector are finite and, in practice, relatively small, which is not the case online. In our case study, approximately 100,000 monitor outcomes are generated for a single hour of observation. This could eventually lead to excessive storage requirements and processing overheads. This potential size problem is addressed through application of statistical SFL, on which our approach relies. It is incremental, so that accumulating counters can be used.

However, another issue is the timeliness of spectrum, for example "is a week-old observation relevant for the current state of the system?" A fault may appear long time after the system was started (e.g., memory leakage, unexpected combination of inputs that affect the internal state of the system, an unnoticed third-party component update). Old spectra might mislead the fault localisation. The detection of a new failure should always lead to the same diagnosis, independent of how long the system has been running.

Note however that the problem is not symmetric, when conversely, a fault is fixed, or the failures are not observed anymore. If the fault is fixed or worked-around knowingly, it is easy to reset the matrix at the same time to avoid this "ageing effect". If the failures stop appearing without the fault having been fixed, it is better to still report the component as faulty for some sufficiently long time to acknowledge the problem and deal with it.

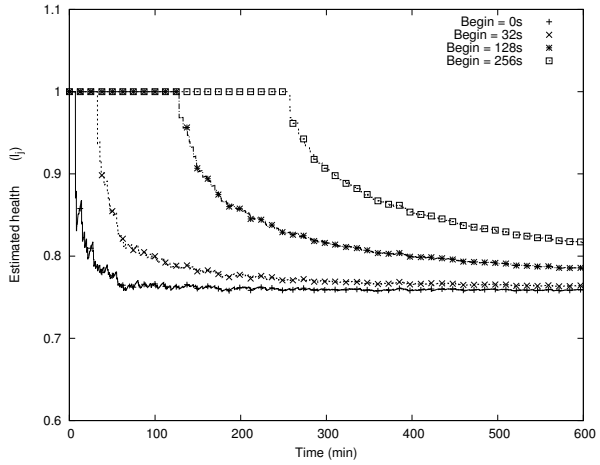

Fig. 6. Estimated health with an infinite window.

Figure 6 shows the health estimated by the SFL algorithm for a faulty component yielding a failure at different times, and we keep all spectra. The later the failure surfaces, the slower is the convergence of health. For a given failure happening at different times, it is better if the algorithm output stays identical independent of how long the system has been running before. The following sub-sections discuss ways to overcome this problem.

1) Sliding Window: A *sliding window* policy discards spectra that are older than a given age.

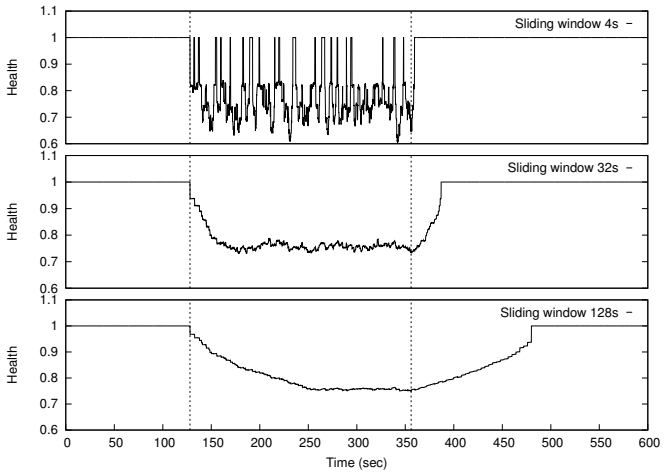

Fig. 7. Estimated health for three sliding window lengths (component fails in the period 128 s – 356 s, dotted lines).

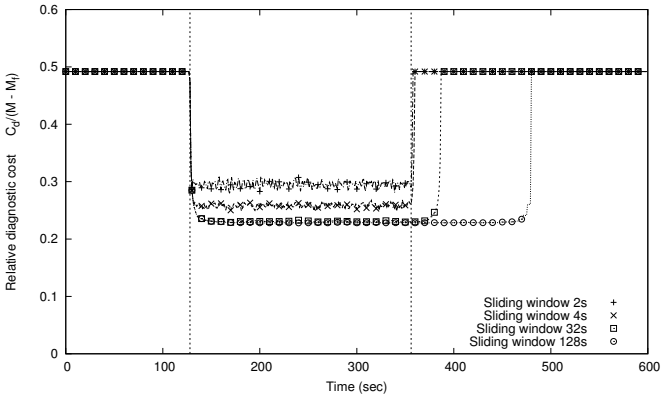

Fig. 8. Average diagnostic cost for four sliding window lengths (component fails in the period 128 s – 356 s, dotted lines).

Figure 7 shows the effect of the sliding window on the estimated health. It depicts only results with the best observation policy from the previous section, but similar results with respect to the window size were observed for every type of policy. Shorter windows lead the SFL to react faster to the detection of a failure, reducing the latency to deal with its corresponding fault. However, short sliding windows result in noisy and fluctuating system health. In the example, a window of 4 s is too short and leads to constant fluctuations in health between 0.6 and 1. The ideal window size (leading to stable health values) depends on the frequency of the monitors generating observations and the frequency of failures being detected.

Figure 8 shows the average diagnostic cost for different window sizes. Short sliding windows ($\leq$4 s) yield a relatively high diagnostic cost and noise, because they are too small to contain enough test outcomes for adequate diagnosis. Longer windows (of size 32 s and more) provide all a similar good diagnostic cost, deprived of fluctuations. They solely differ in the latency to react to the failure disappearance: the longer the window is, the longer is the latency. The size of the

window after which the diagnosis presents no more noise depends on the frequency at which the failures are detected. In the simulations, approximately 60 outcomes per second were generated, with on average 59 *pass* and one *fail*. So an average of a dozen *fail* outcomes per window appears to be sufficient to keep the diagnosis stable.

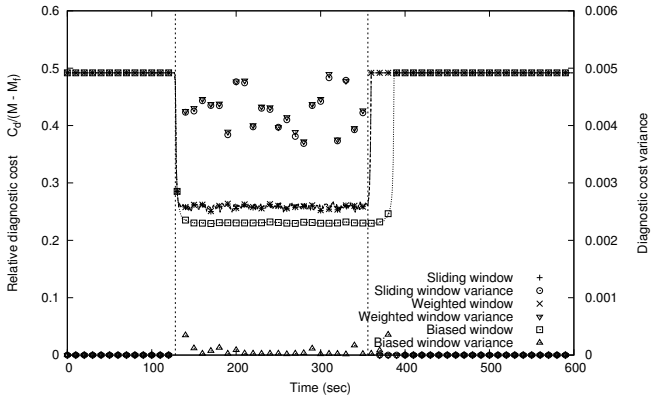

Fig. 9. Average diagnostic cost for the three types of windowing policy with a length of 4 s (component fails in the period 128 s – 356 s, dotted lines).

2) Weighted sliding window: In order to react faster to the appearance or disappearance of faults, a possible technique is to put more weight on the last observations in the computation, while still considering historic observations with smaller weight, in order to avoid the instabilities in the diagnosis. Instead of directly discarding spectra after their window expires, old spectrum data is made less and less relevant by using a weighting window function. Values are weighted exponentially from the oldest (lowest weight) to the most recent one (highest weight).

Figure 9 indicates that using a weighted window of length 4 s has no effect compared to a sliding window. The diagnostic cost is not improved and neither the local variance (computed over the last 32 points). Although this policy allows to have a steeper decrease (resp. increase) of the health of the faulty component when the fault is introduced (resp. removed), the ranking between each component stays mostly unchanged, and therefore the average diagnostic is similar to the simpler policy.

3) Biased sliding window: In cases where failures are very intermittent, discarding spectra containing *fail* outcomes can reduce strongly the quality of the diagnosis because failures provide more diagnostic information than passed spectra. This is the reason for the noise in the average diagnostic cost of small sliding windows.

A biased sliding window policy keeps failing outcomes in the matrix for a longer period of time than passing outcomes. In the experiments, we use a period that is 8 times longer for the *fail* outcomes than for the *pass* outcomes. Figure 9 shows the advantage of this technique as the diagnostic cost is clearly both smaller and more stable. The *pass* outcomes were kept for 4 s while the *fail* outcomes were kept for 32 s. In terms of quality, this provides an equivalent diagnosis to a sliding window that incorporates all the outcomes for 32 s. However it also has the drawbacks of reacting slower to disappearing faults that have been dealt with by the system. So, this policy does not provide any advantages over the simple sliding window policy with a longer window.

Therefore, we recommend the sliding window policy, which is easy to implement and is the fastest to execute. The size of the window should not be too long to ensure a fairly fast reaction in terms of health. It can be set as the minimum duration for which one failure occurrence should affect the diagnosis.

V. Case Study and Evaluation

All techniques for realising online fault diagnosis with SFL have been introduced and developed with the help of synthetic system configurations and simulation. In the following, we evaluate our contributions developed in the previous section.

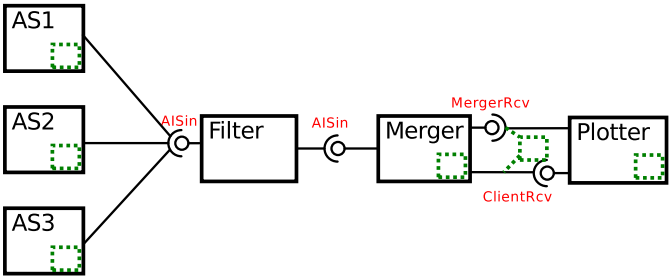

Fig. 10. Architecture of the case study system; monitors are shown as dashed boxes.

A. Surveillance System

The surveillance system that we use as case receives information broadcasts from ships, called *AIS messages* [13], and it processes them in order to form a situational picture of the coastal waters. The (simplified) architecture of this system is displayed in Fig. 10, comprising six main modules. In the real world, the *AS'* modules receive the AIS data from the antennas physically spread along the coast. In the evaluation, they read recorded AIS transmissions, for repeatability of the experiments. The *Filter* module suppresses duplicate messages because some antennas cover overlapping areas. The *Merger* acts as a temporary database of AIS messages, and clients can query it for ship tracks (sailing routes). The *Plotter* is one such client. It displays all ships with their tracks on the screen of the command and control centre (by sending vector drawings to the actual display system). These modules are implemented as Atlas[5] components in Java.

Figure 11 shows an excerpt of the decomposed system, merely for illustrating its complexity on the level of granularity used for monitoring and fault diagnosis (not intended to be read). The system is comprised of 63 components for the core business logic with an average of 10 lines of Java code each.

[5]http://swerl.tudelft.nl/bin/view/Main/Atlas

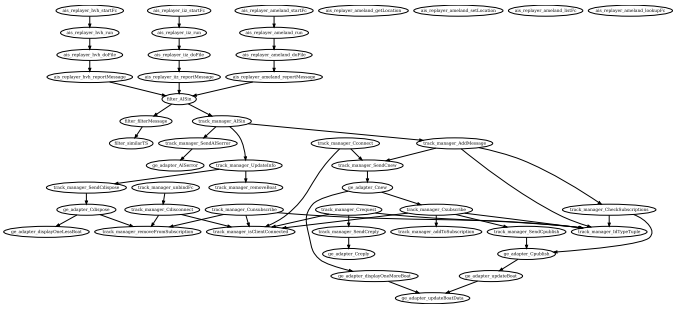

Fig. 11. CIG of the case study on the level of granularity used for monitoring and diagnosis (63 components).

B. Monitoring and Diagnosis Infrastructure

The monitoring infrastructure of our system comprises four monitors, each of them guarding different functional and non-functional aspects of the system.

- **AIS data monitor:** checks that the AIS data sent by ships have valid header and arrive with the correct frequency [13].
- **Track manager monitor:** the behaviour of the track manager is modeled as an FSM, and this monitor verifies that it follows the model.
- **Track manager connection monitor:** the connection between the plotter and the track manager is monitored and error messages are reported as failures.
- **Unhandled exceptions monitor:** watches every component in the system for unhandled Java exceptions. After the exception is detected the component stops.

Coverage of components is recorded through an ad-hoc Java aspect. Every component is augmented with this aspect, which informs a global *involvement manager* whenever a component is covered. This special manager maintains the spectra.

When a monitor generates an outcome, it is first transmitted to the involvement manager to associate the current spectrum, and then passed to another global module called *SFL* performing the online diagnosis described in Section II-A. The output is a list of all components sorted according to their likelihood of containing a fault, plus their associated estimated health (similarity coefficient). The list is updated upon every new monitor outcome, and provides a means to the adaptive and self-managing system to react to unforeseen failures.

Because we did not invest extensive effort in the implementation, the monitors and their logging of the component coverage lead to a significant performance overhead of approximately 50%. This is mainly attributable to the utilisation of an aspect-oriented approach and the high update frequency of the health information, which creates a huge communication overhead in our example system. However, from earlier work, such as in [14], we know that we can bring this overhead down to 5%, mainly by optimizing the spectrum recording and reducing the update frequency. Another option to address the monitoring overhead would be to adapt these mechanisms to work asynchronously, so that they can be partly offloaded to a different processor.

C. Injected Faults

Our industrial partner is interested in two types of faults, hardware faults in the transmission system of local stations, and software faults caused by the business logic.

Hardware faults in the data transmission are simulated through random packet losses. Software faults are introduced through mutations in the original code (a set of 100 mutants was created with μJava [19]). For each of the mutation faults, the system was executed for one hour with the recorded input, producing approximately 100,000 monitor outcomes in total. The output of the SFL component was saved every 10 seconds. A posteriori, it is then possible to determine the diagnostic cost at each moment in time. 12 mutations lead to early system crash (within a minute) and are sorted out (in practice, such bug would be directly noticed and investigated off-line). 55 mutations have faults not detected by the monitors, leaving 33 configurations with detectable faults.

D. Results

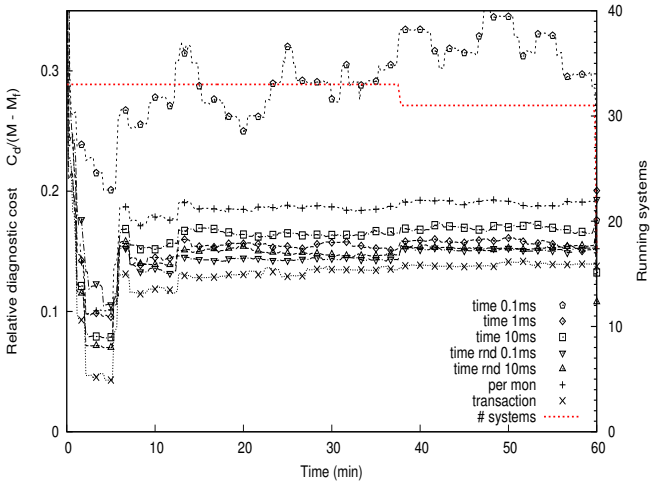

Fig. 12. Average C_d over different periods and for two different observation policies (33 configurations).

The average C_d for *separate per monitor* and *randomized time frame* observation strategies is presented in Figure 12. The SFL algorithm uses a sliding window of 5 minutes, in order to ensure a good quality of the diagnosis while keeping a relatively fast reaction to any fault correction.

The diagnostic cost C_d, which starts at 0.5, decreases until it reaches some relatively constant value after around a minute. This is similar to the results seen in the simulations (Fig. 5). After 5 minutes of execution (i.e., the length of the sliding window), all C_d graphs increase. This is because some faults lead to failures only at initialisation, i.e., they are located in components only used at that time. When these first spectra are removed from the matrix (through the sliding window) the SFL loses information for their location, and assumes a better health, so C_d increases.

As in the simulation, the *separate per monitor* observation performs poorly, with an average C_d converging to 0.19. The

transactional observation performs best, with an average $C_d = 0.14$. The *time frame* observation yields its best results with 1 ms ($C_d = 0.16$). A longer period (10 ms) impairs the results, and even longer periods of 100 ms are devastating, as well as very short periods of 0.1 ms, both leading to C_d around 0.3.

Results suggest that observation periods of 1 ms are optimal for this system. The *randomized time frame* observation performed equally well as the best fixed time period, for all periods tried between 0.1 ms and 100 ms.

E. Discussion

This case study demonstrates the feasibility of online fault localisation using the SFL technique in a real system inspired by industry. With a diagnostic cost ranging on average below 0.2 just after a minute, it also shows that fault localisation is able to point into the right direction for identifying problematic components in adaptive and self-managing systems. Of course, this works only if residual defects can be detected by the monitors. Improving the monitoring of a system is outside the specific scope of this paper. The fact that the results are relatively similar to the results obtained by simulation suggests that the model employed for the simulation is representative for the real case.

Transactional observation provides the best results (in our case); that is, if for each monitor the information about which component is observed is known and correct. Otherwise, a randomized time frame allows diagnosis with comparable quality, if the processing time of a transaction can be approximated and used as observation period.

The example uses a monitoring and diagnosis architecture adapted to systems with a relatively fast connection to a central node. On systems where each component is independent and communication is limited, such as sensor network, a different approach might be necessary. For instance, the coverage spectrum can be sent within the data, instead of using a separate channel, favouring usage of coverage policies like the *transactional* one. As this policy is very efficient, it should be possible to find similar results on such a system.

It is important to discuss some threats to the validity of our study. The main threat to external validity is the usage of only one system for this study. Although the simulations provide a more generic base, they use one basic topology, coming from a distinct application domain. However, questions regarding the performance of SFL w.r.t. system topology and size are subject of current and future work, independent of the main intention of this paper, which is to study and further develop SFL for the online diagnosis context.

A second (internal) threat to the validity of our work to be addressed in the future is the usage of diagnostic cost as indicator of SFL performance, and, thus, as sole indicator for automated fault resolution. Although C_d is used as a standard in the fault diagnosis literature, it is not clear whether it provides a direct indicator on how the recovery process acts with the faulty component ranking, or whether other factors should be considered. In other words, the quality of the diagnosis depends on the recovery process or the self-* process which uses it.

VI. Related Work

The role of fault diagnosis for realising more adaptive, intelligent, and self-aware systems has been recognized for at least a decade (e.g., [1], [2], [3], [12]). Some researchers have looked at online defect detection [6], [20], but did not address the specific issues of finding the root causes of defects, i.e., the diagnosis.

Krämer et al. [16] have presented self-monitoring for a sensor network. As each node is independent from each other, the fault location is straightforward, and can be derived from the observations. Moreover, their approach does not fit more generic systems, apart from sensor networks, and would not detect misbehaviour on the global level.

Seltzer and Small [22] and Chen [8] have proposed system infrastructures for enabling self-monitoring and -adaptation. However, their approaches focus on system performance, ignoring all the other software quality issues, that our approach is able to treat. The biggest drawbacks of these approaches is that they rely on ad-hoc localisation algorithms, which are based on long observations performed in test systems rather than in the operational systems, and that they often require manual adjustments. The usage of automatic diagnosis in our approach avoids these drawbacks. Our approach can be applied in a generic way, and relies only on the latest observations.

Related work in diagnosis follows two main streams. There are (1) spectrum-based approaches such as Pinpoint in the ROC project [7] (focus on dependable web services), and (2) model-based approaches such as in model-based programming [25] where diagnosis as well as recovery are based on logic reasoning in terms of a behavioural model of the system. Due to their inherently high complexity (run-time overhead) and the fact that modeling is costly and error-prone, in particular when systems are dynamically reconfiguring themselves, model-based approaches are limited to systems comprising in the order of hundreds/thousands of simple (hardware) components [10], [26].

Delta Debugging is another related approach [9]. Although it supports larger systems, it requires to record the state of the system at every execution step, which would be prohibitive to perform in an operational system. In [23], an invariant-based approach is presented and applied online. However, they use specialised active unit-testing instead of monitoring, and the state of the system is recorded every time a test is executed, which leads to a very high overhead (execution time multiplied by $\sim$100). An additional issue are interferences that active testing can cause in a running system.

In contrast to these few examples describing applications in an online context, most research has focused on design-time (offline) diagnosis. Spectrum-based approaches are statistical techniques, e.g., Tarantula [15], Ochiai [4], the Nearest Neighbor technique [21], Sober [18], CBI [17], and CrossTab [27]. They are based on similarity coefficients, and are incremental in nature, leading to ultra low space and time complexity,

which makes them ideal candidates to be applied online. This is why our approach uses a statistical technique for diagnosis and the provision of health information.

VII. Conclusions & Future Work

While fault localisation is a fundamental step towards adaptive and self-managing systems, that is for identifying the part of the system "which is to blame", little work so far has focused on adopting existing diagnosis approaches in this domain. In this article, we present an approach for realising online spectrum-based fault localisation to be used in self-adaptive systems. We introduce techniques to obtain a significant spectrum for the SFL algorithm in order to yield good diagnoses. Furthermore, we demonstrate how randomized periodic resetting, in the form of a sliding window, provides high-quality spectra, without having to use specific information about the target system. This results in a diagnostic outcome which is always relevant to the current state of the system. In addition, we show that the usage of a non-weighted time window is sufficient for this purpose.

Our contributions are validated first by simulation of a large set of randomly generated systems, and through a case study with a system inspired by industry. The diagnostic results on a set of mutated systems corroborate the results of the simulation and confirm that, with our contributions, SFL and monitoring can be applied successfully to online fault diagnosis.

Additional challenges could be investigated in future work in order to improve the quality of online diagnosis in real systems. Building an oracle for testing is relatively easy, since inputs, expected outputs, and the components are known. In a live system, producing a monitor with the same degree of accuracy is harder. Imperfections in its design can lead to false positives, i.e., announce a failure while the system is actually behaving correctly. The current SFL diagnosis yields poor results in such cases. This could be improved by enhancing the similarity coefficient algorithm, and by considering in the ranking that monitors may also contain faults.

There are monitors, commonly called "watchdogs", that detect the *disappearance* of a component (lost service). They generate a *fail* outcome when an observed component is not responding. If a component is not covered, it is not included in the spectrum, and cannot be convicted, even though it contains the fault and is the one to blame. In future work, we will investigate how watchdogs can be incorporated into online fault diagnosis.

Acknowledgement

This work has been carried out as part of the Poseidon project under the responsibility of the Embedded Systems Institute (ESI), Eindhoven, The Netherlands. The project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK03021 program.

References

[1] Berkeley/stanford recovery-oriented computing website. http://roc.cs.berkeley.edu/.

[2] NASA model-based diagnosis and recovery website. http://ti.arc.nasa.gov/tech/dash.

[3] XEROX Model-Based Computing project website. http://www2.parc.com/spl/projects/mbc/.

[4] R. Abreu, P. Zoeteweij, and A. van Gemund. On the accuracy of spectrum-based fault localization. In *Proc. TAIC PART'07*, 2007.

[5] R. Abreu, P. Zoeteweij, and A. van Gemund. Spectrum-based multiple fault localization. In *Proc. ASE'09*, 2009.

[6] G. K. Baah, E. Gray, and M. J. Harrold. On-line anomaly detection of deployed software: a statistical machine learning approach. In *In Proc. of the 3rd International Workshop on Software Quality Assurance*, pages 70–77. ACM, 2006.

[7] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the Conference on Dependable Systems and Networks*, pages 595–604, Washington, USA, 2002. IEEE Computer Society.

[8] Z. Chen. Service fault localization using probing technology. In *Proceedings of the Conference on Networking, Sensing and Control*, pages 937–942, Apr. 2006.

[9] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 342–351, New York, USA, May 2005. ACM Press.

[10] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32:97–130, April 1987.

[11] M. J. Harrold and M. L. Soffa. Interprocedual data flow testing. In *TAV3: Proceedings of the third symposium on Software testing, analysis, and verification*, pages 158–167, New York, USA, 1989. ACM.

[12] P. Horn. Autonomic computing: IBM's perspective on the state of information technology. Technical report, IBM, 2001. http://www.research.ibm.com/autonomic/.

[13] International Telecommunication Union. Recommendation ITU-R M.1371-1, 2001.

[14] T. Janssen, R. Abreu, and A. J. C. van Gemund. Zoltar: A toolset for automatic fault localization. In *Proc. ASE'09 - Tool Demonstrations*, 2009.

[15] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proc. ICSE'02*, 2002.

[16] N. Krämer, A. Monger, L. Petrak, C. Hoene, M. Steinmetz, and W. Cheng. A collaborative self-monitoring system for highly reliable wireless sensor networks. In *Proc. of the 2nd IFIP Wireless Days conference*, pages 343–348, Piscataway, USA, 2009. IEEE Press.

[17] B. Liblit. Cooperative debugging with five hundred million test cases. In *Proc. ISSTA'08*, 2008.

[18] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: Statistical model-based bug localization. In *Proc. ESEC/FSE-13*, 2005.

[19] Y.-S. Ma, J. Offutt, and Y. R. Kwon. Mujava: an automated class mutation system: Research articles. *Software Testing Veriication and Reliability*, 15:97–133, June 2005.

[20] C. Rabejac, J.-P. Blanquart, and J.-P. Queille. Executable assertions and timed traces for on-line software error detection. In *Annual Symposium on Fault Tolerant Computing*, pages 138–147, June 1996.

[21] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proc. ASE'03*, 2003.

[22] M. Seltzer and C. Small. Self-monitoring and self-adapting operating systems. In *Proc. of the Workshop on Hot Topics in Operating Systems*, pages 124–129, Washington, USA, 1997. IEEE Computer Society.

[23] D. Slane. Fault localization in in vivo software testing. Master's thesis, Bard College, Massachusetts, USA, 2009.

[24] J. M. Voas. Pie: A dynamic failure-based technique. *IEEE TSE*, 18(8):717–727, 1992.

[25] B. C. Williams, M. D. Ingham, S. H. Chung, and P. H. Elliott. Model-based programming of intelligent embedded systems and robotic space explorers. In *In Proceedings of the IEEE: Special Issue on Modeling and Design of Embedded Software*, pages 212–237, 2003.

[26] B. C. Williams and R. J. Ragno. Conflict-directed a* and its role in model-based embedded systems. *Discrete Appl. Math.*, 155:1562–1595, June 2007.

[27] W. Wong, T. Wei, Y. Qi, and L. Zhao. A crosstab-based statistical method for effective fault localization. In *Proc. ICST'08*, 2008.

[28] Y. Wu, D. Pan, and M.-H. Chen. Techniques for testing component-based software. In *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems*, pages 222–232, Los Alamitos, USA, 2001. IEEE Computer Society.