Diagnosing architectural run-time failures

Paulo Casanova, David Garlan, Bradley Schmerl
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{paulo.casanova|schmerl|garlan}@cs.cmu.edu

Rui Abreu
Department of Informatics Engineering
Faculty of Engineering of University of Porto
Porto, Portugal
rui@computer.org

***Abstract*—Self-diagnosis is a fundamental capability of self-adaptive systems. In order to recover from faults, systems need to know which part is responsible for the incorrect behavior. In previous work we showed how to apply a design-time diagnosis technique at run time to identify faults at the architectural level of a system. Our contributions address three major shortcomings of our previous work: 1) we present an expressive, hierarchical language to describe system behavior that can be used to diagnose when a system is behaving different to expectation; the hierarchical language facilitates mapping low level system events to architecture level events; 2) we provide an automatic way to determine how much data to collect before an accurate diagnosis can produced; and 3) we develop a technique that allows the detection of correlated faults between components. Our results are validated experimentally by injecting several failures in a system and accurately diagnosing them using our algorithm.**

I. Introduction

Within a self-adaptive system, fault diagnosis and localization is one of the most important concerns – corresponding to the Monitoring and Analysis parts of the "classical" MAPE loop [15]. Automated fault diagnosis is necessary for recognizing when a system needs to adapt to problems. And fault localization is important to focus the adaptation mechanism on the parts of the system that need attention.

In our own research on architecture-based self adaptation we have proposed the use of a technique termed Spectrum-based Multiple Fault Localization (SMFL) [4], which provides a list of candidate fault explanations, ranked by probability of likelihood in causing a detected problem [6]. The key idea behind the technique is to identify finite transactions of run-time behavior, and the sets of architectural elements that were involved in those behaviors. As we describe in more detail later, each of these transactions is evaluated using predicates that judge success or failure of the behavior based on properties of the transaction. A collection – or window – of such transactions can then be analyzed to determine which architectural element, or sets of architectural elements, could have caused the observed successes and failures.

SMFL was originally developed for development-time debugging, where each generated transaction is the result of running a test case. In our research we have shown how to adapt the ideas to the run-time setting using architectural (component and connector) models. Specifically, in [6] we proposed a simple specification language, based on Message Sequence Charts (MSCs), for specifying types of behaviors to monitor, which we termed *transaction types*, and the use of predicates over instances of those types as oracles for judging success and failure. We were able to show that this approach could be effective in identifying faults, and demonstrated its use in the context of typical web-based applications, such as news servers.

The runtime infrastructure for diagnosis is illustrated in Figure 1. We probe the *Base System* to observe low-level events of interest. For this, we can use a variety of off-the-shelf techniques such as aspects, standard network-based monitoring tools, or wrappers around system calls. Probed events are fed into the *Recognizer*, which matches these events against the set of transaction types as described by a *Behavior Model*, which specifies what patterns of events should be recognized and how they are related to elements in the architecture, to produce transactions. The *Oracle* takes each transaction and determines whether or not it was successful, using the *Correctness Criteria*. All transactions are passed to the *Fault Localizer*, which periodically analyzes a subset of them using the STACCATO [1], [2] algorithm to determine the health of each element in the transaction. The *Fault Localizer* then reports the health of each element involved in the transaction, which can be used by a self-adaptive system to plan adaptations to repair unhealthy elements.

The results of this work were encouraging in that we were able to detect and localize many common faults with good performance. Further experience has identified a number of limitations of our initial approach. First is the *choice of language for specifying behaviors*. While MSCs, are simple and relatively intuitive for specifying finite behaviors, they have several serious limitations including their expressiveness and potential for reuse. In terms of expressiveness, there are some kinds of behaviors that are difficult, if not impossible, to specify. For instance, MSCs are not capable of specifying that an event *does not happen*. This is a crucial limitation, because it makes it difficult to identify faults where non-response is a possibility, and should be noticed since it may reflect the failure of a component. Moreover, as we explain later, MSCs do not provide appropriate support for reuse: for example, because they do not provide any support for specialization, two MSCs which differ only in a single message will have to be fully coded as separated MSCs.

The second limitation is the lack of a principled way to determine the window size for collecting transactions. As we elaborate later, selecting an appropriate window is crucial for a successful application of the technique: if the window is too small, there is not enough evidence to generate an accurate diagnosis; but if the window is too large, there may be a long delay between the occurrence of a problem and

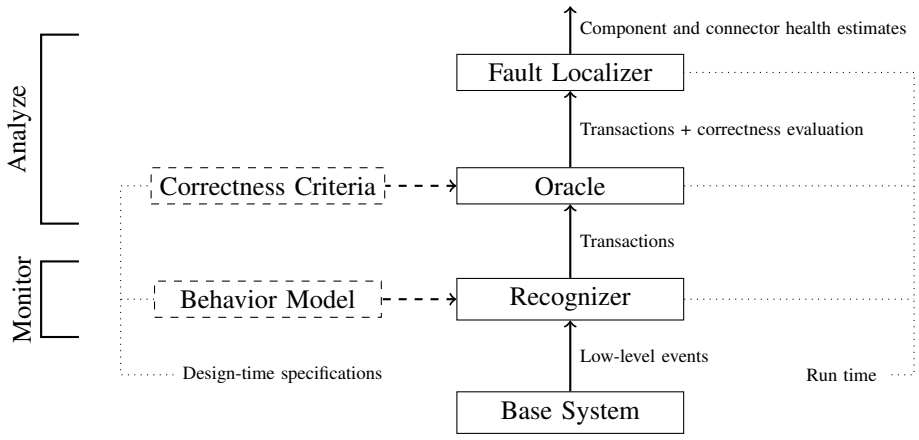

Fig. 1. Approach overview.

its diagnosis. In our earlier work we used experimentation to determine an appropriate window. However, this has its own limitations. Most importantly, it does not allow one to vary the window dynamically, for example when more transactions are generated under high load. Also, it requires a large amount of up-front calibration, raising the cost of using the method.

The third limitation is the inability to detect *correlated faults*. While SMFL is specifically designed to identify situations where multiple faults may occur, it assumes that those faults are independent (i.e, not correlated). In practice, however, faults are often correlated. For example, a particular server may only fail when accessing a particular database.

In this paper we show how to address these three limitations. With respect to behavior specification, we introduce a language, based on first-order predicate logic, that has more expressiveness than MSCs, and also allows one to reuse behavior patterns by refining abstract patterns to concrete ones, using a form of behavior specialization – providing a form of inheritance that simplifies the specification of specific behavior patterns for a new system. With respect to window size, we introduce the concept of *entropy* as a run-time measure of adequacy for window size. The basic idea is to continue to collect more transactions until the probabilities of candidate explanations for the fault rise to a significant enough level. With respect to correlated faults, we show how SMFL can be adapted, with some loss of performance, to handle those situations. For all three techniques, we illustrate their use in terms of a common running example, and provide experimental evidence of their usefulness.

The remainder of this paper is organized as follows: in Section II we summarize the SMFL algorithm as applied to testing, and introduce an example that will be used for applying and evaluating our approach a run time. We present our approach in Section III, highlighting the improvements in our technique over our previous approach. Our evaluation is described in Section IV, and how our work is positioned with respect to related work in V. Finally, in Section VI, we discuss future work.

II. Background

In this section we summarize the reasoning approach to fault localization considered in this paper and introduce the the system used throughout this paper.

A. Classical SFML

Fault localization based on reasoning over program spectra is characterized by the use of (a) *program spectra*, [1] abstracting from actual observation variables, structure, and component behavior; (b) a low-cost, heuristic reasoning algorithm, STACCATO [2], [1], to extract the significant set of *multiple-fault candidates*; and (c) abstract, intermittent models, that take into account that a faulty component (or combination of components) may behave correctly with a specific probability, to compute the *candidate probability* of being the true fault.

In this section, we describe how SFML works, defining how spectra are denoted, and how candidate faulty elements are generated and ranked.

1) Program Spectra: Assume that a software system is comprised of a set of M components c_j where $j \in \{1,\ldots,M\}$, and can have multiple faults, the number being denoted C (fault cardinality). A *diagnostic report* $D =< \ldots, d_k, \ldots >$ is an ordered set of diagnostic (possibly multiple-fault) candidates, d_k, ordered in terms of likelihood to be the true diagnosis.

A program spectrum is a collection of flags indicating which components have been *involved* in a particular dynamic behavior of a system. Our behavioral model is represented simply by a set of components *involved* in a computation, and does not have to indicate at a detailed behavioral level exactly what that involvement was. Thus, recording program spectra is light-weight, compared to other run-time methods for analyzing dynamic behavior (e.g., dynamic slicing [18]). Although we work with these so-called component-hit spectra, the approach outlined in this section easily generalizes to other types of program spectra [14], possibly with additional overhead in time or space.

Program spectra are collected for N (pass/fail) executions of the system. Both spectra and program pass/fail information

[1] *Spectra* used in classical SMFL refers, in the scope of our work, to the set of architectural elements that exist in a *transaction*.

are input to spectrum-based fault localization. The program spectra are expressed in terms of a $N \times M$ *activity matrix* A. Table I illustrates a small (A,e) pair for 3 components and 4 observations. An element a_{ij} has the value 1 if component j was observed to be involved in the execution of run i, and 0 otherwise.

2) Candidate Generation: As with any model-based diagnosis (MBD) approach, the basis for fault diagnosis is a model of the program. Unlike many MBD approaches, however, no detailed modeling is used, but rather a generic component model. Each component (c_j) is modeled in terms of the logical proposition

$$h_j \Rightarrow (ok_{inp_j} \Rightarrow ok_{out_j}) \tag{1}$$

where the booleans h_j, ok_{inp_j}, and ok_{out_j} model component health, and the (value) correctness of the component's input and output variables, respectively. "Correctness" is broadly defined including evaluation of quality attributes in addition to normal functional correctness. The above *weak* model[2] specifies nominal (required) behavior: when the component is correct ($h_j = \mathsf{true}$) and its inputs are correct ($ok_{inp_j} = \mathsf{true}$), then the outputs must be correct ($ok_{out_j} = \mathsf{true}$). As Eq. (1) only specifies nominal behavior, even when the component is faulty and/or the input values are incorrect it is still possible that the component delivers a correct output. Hence, a program pass does not imply correctness of the components involved.

c_1	c_2	c_3	e	
1	1	0	1	obs_1
0	1	1	1	obs_2
1	0	0	1	obs_3
1	0	1	0	obs_4

TABLE I. PROGRAM SPECTRA EXAMPLE

By instantiating the above equation for each component involved in a particular run (row in A) a set of logical propositions is formed. Since the input variables of each test can be assumed to be correct, and since the output correctness of the final component in the invocation chain is given by e (pass implies correct, fail implies incorrect), we can logically infer component health information from each row in (A,e). To illustrate how candidate generation works, for the program spectra in Table I we obtain the following health propositions for h_j:

$$\begin{aligned} \neg h_1 \vee \neg h_2 \quad & (c_1 \text{ and/or } c_2 \text{ faulty}) \\ \neg h_2 \vee \neg h_3 \quad & (c_2 \text{ and/or } c_3 \text{ faulty}) \\ \neg h_1 \quad & (c_1 \text{ faulty}) \end{aligned}$$

These health propositions have a direct correspondence with the original matrix structure: there is one line for each failing run and the boolean elements in each line correspond to the components that participated in the observation. Note that only failing runs lead to corresponding health propositions, since (because of the conservative, weak component model) from a passing run no additional health information can be inferred.

As in most MBD approaches, the health propositions are subsequently combined to yield a diagnosis by computing the so-called minimal hitting sets (MHS, minimal set cover), i.e., the minimal health propositions that cover the above propositions. In our example, candidate generation yields two double-fault candidates $d_1 = \{1,2\}$, and d_2 = $\{1,3\}$. The step of transforming health propositions into diagnosis is generally responsible for the prohibitive cost of reasoning approaches. However, we use an ultra-low-cost heuristic MHS algorithm called STACCATO [2], [1] to extract only the significant set of multiple-fault candidates d_k, avoiding needless generation of a possibly exponential number of diagnostic candidates. This allows a spectrum-based reasoning approach to scale to real-world programs [4].

3) Candidate Ranking: The previous phase returns diagnosis candidates d_k that are logically consistent with the observations. However, despite the reduction of the candidate space, the number of remaining candidates d_k is typically large, not all of them equally probable. Hence, the computation of diagnosis candidate probabilities $\Pr(d_k)$ to establish a *ranking* is critical to the diagnostic performance of reasoning approaches. The probability that a diagnosis candidate is the actual diagnosis is computed using Bayes' rule, that updates the probability of a particular candidate d_k given new observational evidence (from a new observed spectrum).

The Bayesian probability update, in fact, can be seen as the foundation for the derivation of diagnostic candidates in any reasoning approach: i.e., (1) deducing whether a candidate diagnosis d_k is consistent with the observations, and (2) computing the posterior probability $\Pr(d_k)$ of that candidate being the actual diagnosis. Rather than computing $\Pr(d_k)$ for *all* possible candidates, just to find that most of them have $\Pr(d_k) = 0$, candidate generation algorithms are used as shown before, but the Bayesian framework remains the formal basis.

For each diagnosis candidate d_k the probability that it describes the actual system fault state depends on the extent to which d_k explains all observations. To compute the posterior probability that d_k is the true diagnosis given observation obs_i (obs_i refers to the coverage and error information for computation i) Bayes' rule is used:

$$\Pr(d_k|obs_i) = \frac{\Pr(obs_i|d_k)}{\Pr(obs_i)} \cdot \Pr(d_k|obs_{i-1}) \tag{2}$$

The denominator $\Pr(obs_i)$ is a normalizing term that is identical for all d_k and thus need not be computed directly. $\Pr(d_k|obs_{i-1})$ is the prior probability of d_k. In the absence of any observation, $\Pr(d_k|obs_{i-1})$ defaults to $\Pr(d_k) = p^{|d_k|} \cdot (1-p)^{M-|d_k|}$, where p denotes the *a priori* probability that component c_j is at fault, which in practice we set to $p_j = p$. $\Pr(obs_i|d_k)$ is defined as

$$\Pr(obs_i|d_k) = \begin{cases} 0 & \text{if } obs_i \wedge d_k \text{ are inconsistent;} \\ 1 & \text{if } obs_i \text{ is unique to } d_k; \\ \varepsilon & \text{otherwise.} \end{cases} \tag{3}$$

As mentioned earlier, only candidates derived from the candidate generation algorithm are updated, meaning that the 0-clause need not be considered in practice.

[2]Within the model-based diagnosis community, two broad categories of model types have been specified: (1) weak-fault models, which describe a system only in terms of its normal, non-faulty behavior, and (2) strong-fault models, which also include a definition of some aspects of abnormal behavior.

In model-based reasoning, many policies exist for defining ε [9]. Amongst the best ε policies is one that uses an *intermittent* component failure model, extending h_j's permanent, binary definition to $h_j \in [0,1]$, where h_j expresses the probability that faulty component j produces correct output. ($h_j = 0$ means persistently failing, and $h_j = 1$ means healthy, i.e., never inducing failures).

Given the intermittency model, for an observation $obs_i = (A_{i*}, e_i)$, the ε policy in Eq. (3) becomes

$$\varepsilon = \begin{cases} \prod_{j \in d_k \wedge a_{ij}=1} h_j & \text{if } e_i = 0 \\ 1 - \prod_{j \in d_k \wedge a_{ij}=1} h_j & \text{if } e_i = 1 \end{cases} \quad (4)$$

Eq. (4) follows from the fact that the probability that a run passes is the product of the probability that each involved faulty component exhibits correct behavior. (Here we adopt an *or*-model; we assume components fail independently, a standard assumption in fault diagnosis for tractability reasons which we will extend later in section III-C.)

Before computing $\Pr(d_k)$ the h_j must be estimated from (A, e). There are several approaches that approximate h_j by computing the probability that the *combination* of components involved in a particular d_k produce a failure, instead of computing the *individual* component intermittency rate values [3], [10]. Although such approaches already give significant improvement over the classical model-based reasoning (see [4] for results), more accurate results can be achieved if the individual h_j can be determined by an exact estimator. To compute such an estimator, h_j is determined per component based on their effect on the ε policy (Eq. (4)) to compute $\Pr(d_k)$. The key idea is to compute the h_js for the candidate's d_k faulty components that *maximizes the probability* $\Pr(obs|d_k)$ *of a set of observations obs occurring, conditioned on that candidate* d_k (maximum likelihood estimation for naïve Bayes classifier d_k). Hence, h_j is solved by maximizing $\Pr(obs|d_k)$ under the above epsilon policy, according to $\arg\max_{\{h_j \mid j \in d_k\}} \Pr(obs|d_k)$.

To illustrate how candidates are ranked, consider the computation of $\Pr(d_1)$. As the four observations are independent, from Eq. (3) and Eq. (4) it follows

$$\Pr(obs|d_1) = (1 - h_1 \cdot h_2) \cdot (1 - h_2) \cdot (1 - h_1) \cdot h_1 \quad (5)$$

Assuming candidate d_1 is the actual diagnosis, the corresponding h_j are determined by maximum likelihood estimation, i.e., maximizing Eq. (5). For d_1 it follows that $h_1 = 0.47$ and $h_2 = 0.19$ yielding $\Pr(obs|d_1) = 0.185$ (note, that c_2 has much lower health than c_1 as c_2 is not exonerated in the last matrix row, in contrast to c_1). Applying the same procedure for d_2 yields $\Pr(obs|d_2) = 0.036$ (with corresponding $h_1 = 0.41$, $h_3 = 0.50$). Assuming both candidates have equal prior probability p^2 (both are double-fault candidates) and applying Eq. (2) it follows $\Pr(d_1|obs) = 0.185 \cdot p^2 / \Pr(obs)$ and $\Pr(d_2|obs) = 0.036 \cdot p^2 / \Pr(obs)$. After normalization it follows that $\Pr(d_1|obs) = 0.84$ and $\Pr(d_2|obs) = 0.16$. Consequently, the ranked diagnosis is given by $D =< \{1,2\}, \{1,3\} >$.

B. The ZNN example

To illustrate and evaluate our approach, we use a custom-built web system, znn. Znn is a typical web system using a standard LAMP stack (Linux, Apache, MySQL, PHP) mimicking a news site with multimedia new articles. Znn's architecture is depicted in Figure 2.

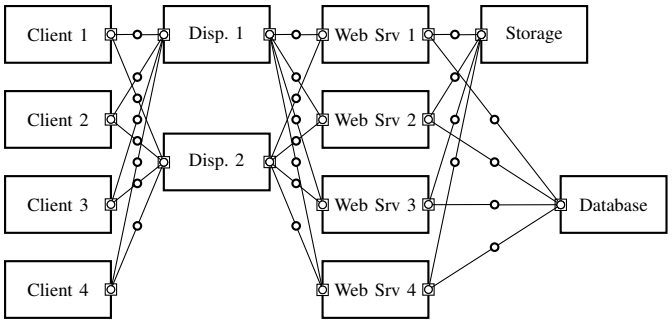

Fig. 2. Architecture of the znn web system used for evaluation.

In this system, multiple clients access one of two dispatchers (also termed "load balancers"), which forward requests to a random web server in a farm. If the request is not for an image, the web server will access the database to fetch the required information and generate the news page with HTML text and references to images. Web clients will then access the system to fetch the images. Images are served from a separate file system storage component, shared among all web servers.

To provide observations about the system to the diagnosis infrastructure, we attach probes to the dispatchers and web servers. These probes report low-level system calls detected using Linux's *ptrace* mechanism such as OPEN(2) or BIND(2). The probes relay events to the *Recognizer* in figure 1.

III. Approach

This section details our approach to perform automatic diagnosis in a running system.

A. Describing architecture-level behavior

As mentioned in Section II-B, the probes placed in znn provide low level system events. However, system architects reason about system behavior at the component and connector level. For znn, they would reason about architectural concepts such as dynamic web requests (to serve news pages) and static web requests (to serve images). In our previous work, we assumed we could identify those requests directly from observable events using MSCs. As we will see in more detail this section, MSCs have two important limitations: they do not allow reasoning about events that do not happen and they do not allow reuse of specifications among similar, but different, systems. But because they do provide an intuitive way of specifying behavior *as long as the behavior conforms to certain constraints*, we still support them as surface syntax for the more expressive language that is described below.

Identifying architecture-level transactions directly using low-level events is possible but it is not intuitive, non-portable and hard to maintain: a small change in the probing system (for example, porting znn to a different platform) could require that all system behaviors be rewritten to use the new low-level events of the new platform.

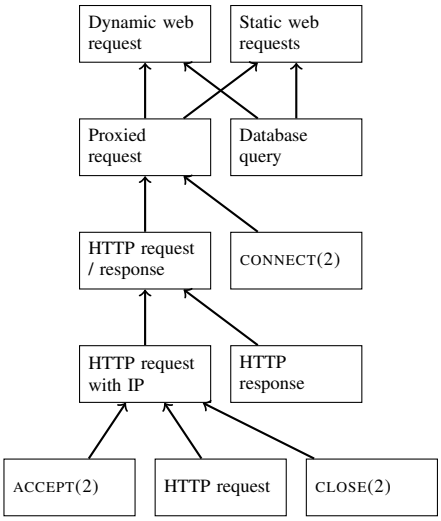

Fig. 3. Partial hierarchy of transactions used in the znn example.

To address this problem, we decompose the system behavior into a hierarchy in which detection of lower-level transactions feeds events for the detection of higher-level transactions. Figure 3 contains a partial hierarchy of events and transactions. The higher-level transactions (dynamic web request and static web requests) can be computed from the proxied requests – HTTP requests sent to the load balancer and forwarded to a web server – and database queries.

```
1  // Defining of HTTP result codes.
2  typedef int32 http_result {
3      invariant self >= 0 and self <= 599;
4  }
5
6  // Proxied request: send from client to dispatcher and
7  // then to a web server (and then back to the client).
8  computation type pxr : tc::htc{
9      result : http_result;
10 }
11
12 // Database query.
13 computation type dbq : tc::htc{}
14
15 // Static and dynamic web requests.
16 computation type dwr : tc::htc{
17     result : http_result;
18     dwr(r : http_result) {
19         result = r;
20     }
21 }
22 computation type swr : tc::htc{
23     result : http_result;
24     swr(r : http_result) {
25         result = r;
26     }
27 }
```

Listing 1. Simplified definition of dynamic and dynamic web requests.

In listing 1 four simplified types of transactions (declared as COMPUTATION TYPE) are defined: PXR, a proxied request, DBQ, a database query issued from a web server, DWR, a dynamic web requests and SWR, a static web request. All these four computations inherit properties from the generic HTC (host/thread computation) in "package" TC (threaded computation). The threaded computation package defines the concept of a thread running in a host.

The language contains primitives akin to well-known object-oriented languages. This has the advantage of easing the learning curve for software engineers and bringing the power of established development techniques. For example, *computation types* are akin to *classes* and *families* to *namespaces* (or *packages*). Computation types represent *events* in the system, either detected by probes or fired by recognition of transactions.

Recognizing transactions from computations (events reported by probes or transactions already identified) is done through the definition of *recognizers* as shown, in simplified form, in Listing 2. The detection of these high-level transactions demonstrates the need for more expressiveness than what MSCs could give us in [6]. Static and dynamic web requests are, essentially, equal, except that dynamic web requests involve a database query whereas static web requests do not. In our recognition language, we can now express the difference in the two transactions.

```
1  // The dwr recognizer
2  recognizer dwr_rgn(r : pxr) {
3    invariant exists q : dbq | r->during_same_thread(q);
4    emit new dwr(r.result);
5  }
6
7  // The swr recognizer
8  recognizer swr_rgn(r : pxr) {
9    invariant not (exists q : dbq | r->during_same_thread(q));
10   emit new swr(r.result);
11 }
```

Listing 2. Identifying dynamic web requests and static web requests.

The two recognizers in Listing 2 identify the static and dynamic web requests from proxied requests depending on whether a database query is made during the process of the request or not. The INVARIANT clause contains the first-order logic condition under which the transaction is recognized. The EMIT clause defines which transaction is identified and initializes it with the data from the lower-level computations.

Also, in Listing 2 we can see the other limitation of MSCs. In plain English, a proxied request is a dynamic web request when *there is at least one database query performed during the request by the same thread*. The DURING_SAME_THREAD method of the PXR computation type is actually defined in its super type, the TC::HTC computation type. This method is defined as in Listing 3.

```
1  family tc {
2      computation type htc {
3          // ...
4          bool during_same_thread(c : htc) {
5              return same_thread(c)
6                  and start(c) >= start(this)
7                  and end(c) <= end(this);
8          }
9          // ...
10     }
11 }
```

Listing 3. Definition of the DURING_SAME_THREAD method.

The sort of reuse of connection logic provided in Listing 3 is not possible to achieve using MSCs. The separation of behavior into different families and structures also allows much easier understanding and reuse. The transactions we recognize in listing 2 can be used for any system that proxies requests to web servers regardless of: (1) how PXR and DBQ are detected and (2) whether DWR and SWR are the most high-level transactions specified or whether other transactions are defined on top of those.

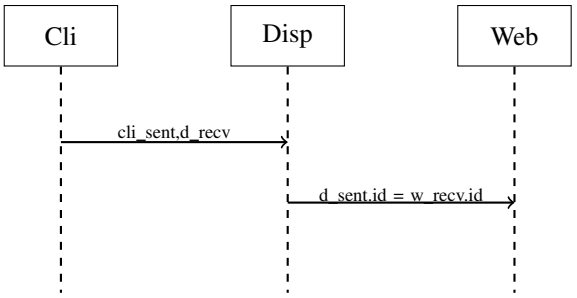

Fig. 4. Message sequence diagram recognizing a computation.

As stated in the introduction of this section, use of MSCs is still possible as they can be converted into recognizers with the following assumptions:

- Each message in the MSC corresponds to two lower-level events: a message sent from the origin and a message arrive at the destination;
- All events detected in an architectural element are detected in the same thread;
- All events inherit from the TC::HTC computation type.

As an example of the application of this transformation, consider the message sequence diagram in Figure 4. In this small example, the client sends a request to the dispatcher which forwards it to the web server. There are three observable events: the request is received at the dispatcher (D_RECV), the request is sent from the dispatcher (D_SENT) and the request is received at the web server (W_RECV). D_SENT.ID = W_RECV.ID is the condition used to link the two events (in this case we use a unique ID sent with every request).

Listing 4 contains the transformation of the MSC into a recognizer with the assumptions stated earlier. With transformation rules like these we can, under the previously stated assumptions, automatically convert MSCs into recognizers. This allows software designers to express the behavior of the system using a more intuitive notation as long as the description fits into the limitations of the MSCs.

```
1  recognizer my_rgn(c1 : d_recv, c2 : d_sent, c3 : w_recv) {
2    invariant
3      // Linkage between d_recv and d_sent.
4      // Note that this is an instantiation of template
5      // code which is always the same for two
6      // events in the same thread.
7      c1->same_thread(c2)
8      and start(c2) >= end(c1)
9      and not (exists c2b : d_sent | c1->same_thread(c2b)
10       and start(c2b) >= end(c1)
11       and start(c2b) < start(c2))
12     // Linkage between d_sent and w_recv.
13     // Note that this is an instantiation of a template
14     // that makes use of the MSC condition.
15     and start(c3) >= end(c2)
16     and c3.id == c2.id
17     and not (exists c3b : w_recv | start(c3b) >= end(c2)
18       and c3b.id == c2.id
19       and start(c3b) < start(c3));
20   emit new proxied_http_request(r.result);
21 }
```

Listing 4. Result of transformation of the MSC.

B. Defining a window size

As previously presented, SFML requires *program spectra* – the (A,e) of section II-A – which, with our approach, is collected at run time. In our previous work we used a fixed time window to collect transactions whose size was determined experimentally. Making the time window fixed in size has several drawbacks: it requires a significant amount of up-front calibration and it does not allow for dynamic adaptation to system changes. If the time window is set too large then problems may be underestimated due to the high volume out-of-date transactions that may skew the diagnosis towards past behavior [6]. If the time window is set too small then diagnosis results may be inconclusive.

There are a number of criteria that might be used to determine this window. Based on the previous discussion and our work in [6], we decided on the following two elementary requirements:

- It has to be large enough to produce a good diagnosis: if not enough evidence is collected, diagnosis may be inconclusive;
- It must be as small as possible to discard out-of-date past transactions and obtain an accurate diagnosis as fast as possible: if a component fails at a certain point in time, all previous successful transactions will only reduce the confidence on the diagnosis;

Since our main goal is diagnosis accuracy and accuracy is highly related to the information collected in the program spectra, we need a metric that computes whether a spectra has *enough information* for diagnosis. As explained before, SFML explores a Bayesian update framework (namely, Barinel [2], [4]) that determines the set of valid diagnosis candidates and assigns probabilities to them. Given that the diagnostic ranking is a list of candidates, in order of likelihood of being faulty, one can compute the entropy of the ranking as follows

$$\mathrm{H}(D) = -\sum_{d_k \in D} \Pr(d_k) \times \log(\Pr(d_k))$$

The entropy (borrowed from Information Theory [24]) characterizes the (im)purity of an arbitrary collection of, in our case, diagnosis candidates. The idea is to adapt the time window considered given the entropy, knowing that more valid informations decrease the entropy of the ranking.

Therefore, depending on the rate at which transactions are generated, this window will change as the system runs. Our implementation defines two parameters: a time rate parameter Δ and a maximum entropy H_m. We perform diagnosis every Δ and our window W initially is set to Δ.

At regular intervals of Δ we apply SFML to all transactions that completed up to W. SFML produces a set of candidates d_k ranked probabilistically ($\Pr(d_k)$) and we compute the entropy of the distribution, $\mathrm{H}(D)$.

If $\mathrm{H}(D) \leq H_m$ then we consider the diagnosis to be accurate and we output the result of the diagnosis. If $\mathrm{H}(D) > H_m$ then we need to collect more data and increase W by Δ. This means that the next time we apply the SFML algorithm, we will use

all the data we have plus all transactions that finished in the last Δ.

We reset the time window to Δ as soon as we produce a diagnosis result to start collecting data for the next diagnosis. Because W increases with Δ and we compute the diagnosis every Δ, past information is prevented from interfering in future diagnosis.

C. Adapting SMFL to detect correlated faults

In general, when software errors are detected, several architectural elements may have been involved in the execution of the computation. In some of those cases the fault may be attributed to one of those components. But in other cases, the fault is a *correlated fault*: it results from the interaction of multiple components and connectors and is unrelated to individual failures. For example, a difference in the output guarantees of a component and input preconditions of another may yield a failure with both components being perfectly healthy on their own. Another example is when the connection between two components is faulty and may introduce errors in the communication.

SMFL assumes each component c_j has a health indicator, $h_j \in [0,1]$, which represents the probability of the component generating a correct output given a correct input. When two components c_i and c_j are invoked, the probability of success is given by $h_i h_j$: a model that assumes that failures are independent and, therefore, the failure of one component is not related to the failure of other components.

A simple example illustrates this: if c_i and c_j always fail when used together and if 50% of the time each one of them is used, it is used with the other one, the standard SMFL would report $h_i < 1$ with 50% probability and $h_j < 1$ with 50% probability. This result means it is equally likely that either c_i or c_j are faulty. However, we would like the algorithm to report that, with 100% probability, failures occur when *both* components are used.

In order to extend SMFL to support correlated faults, we add virtual components representing the interactions among the various components. If c_i and c_j represent two components with health h_i and h_j then $c_{i,j}$ represents the interaction between c_i and c_j and $h_{i,j}$ its health.

Each spectrum is extended with all interactions. The example from Table I would become Table II under this new model.

c_1	c_2	c_3	$c_{1,2}$	$c_{1,3}$	$c_{2,3}$	e	
1	1	0	1	0	0	1	obs_1
0	1	1	0	0	1	1	obs_2
1	0	0	0	0	0	1	obs_3
1	0	1	0	1	0	0	obs_4

TABLE II. EXTENDED PROGRAM SPECTRA EXAMPLE

This process increases the number of "components" that SMFL has to handle. If the system has N components, in order to detect correlated faults of 2 components, $O(N^2)$ virtual components have to be added. In order to detect correlated faults of 3 components, $O(N^3)$ virtual components have to added. The total number of components SMFL has to handle is $O(N^F)$ where F is the maximum number of components SFML has to handle if we want to handle *all* possible combinations of correlated faults.

Adding virtual components increases the computational cost but does not affect the correctness of the diagnosis result. If c_i is faulty then adding $c_{i,j}$ is inconsequential. Similarly, if the problem is not in c_i or c_j but actually in $c_{i,j}$, this will be correctly identified by SMFL. These results are guaranteed by SMFL's optimality theorem described in [4].

Pinpointing the diagnosis result to $c_{i,j}$ does not mean that the original problem source might not be in c_i or c_j individually. After all, c_i and c_j are the only "real" components so the *bug* is likely to be in either. But the standard SMFL approach will generally give results with less confidence if $c_{i,j}$ is not considered as previously shown. With the virtual components inserted, it will correctly pinpoint $h_{i,j}$ as the single source of the problem.

IV. EVALUATION

In this section, we evaluate the diagnostic capabilities and efficiency of the proposed approach using the znn example.

A. Evaluation scenario

To illustrate how our approach can detect both functional and quality of service problems in a system, we injected five different fault scenarios into znn that manifest themselves in different ways:

- Functional failure: an image was not found in the storage.
- Functional failure: a web server has a bug and is not able to find the image to serve.
- Performance failure: a web server's response time degrades.
- Security failure: a client attempts a denial-of-service attack.
- Performance failure (correlated): a web server is slow to respond but only when requests come from a specific dispatcher.

Detecting these failures requires some correctness criteria to be defined. We define the following criteria:

- HTML response codes in the range 4xx and 5xx represent failures.
- Response times above 2 seconds represent failures.
- Client request rates over 1 request / second for at least 5 seconds represents a failure.

Note that repair is not in the scope of our work: we are solely concerned with pinpointing the failed component. Strategies such as checking the image directory for consistency, rebooting a slow server or blocking a malicious client would be handled by later stages in the MAPE loop, which would use input from the diagnosis.

The experiment demonstrates that our diagnosis system is able to: (1) identify each of the faults correctly, and (2) identify multiple, correlated, faults when applicable.

B. Designing recognizers and oracles

We designed the recognizer as described in Section III-A by intercepting the following system calls: `socket(2)`, `bind(2)`, `accept(2)`, `close(2)` `read(2)`, `write(2)` and `connect(2)`. We had to track several other system calls such as `clone(2)` to keep track of the process IDs and thread IDs and we had to track other system calls that can be mapped to the ones above such as `accept4(2)`, `readv(2)` and `writev(2)`. Using these, we built recognizers for the following higher-level computations shown in Figure 3.

We detect correctness as we did in our previous work by defining predicates over transactions which are evaluated by an *Oracle* as presented in figure 1. In our language, we can define several oracles: each transaction is evaluated in all oracles that are applicable and is considered a success if and only if all oracles evaluate it as a success. As an example, the following listing contains the oracle that states that request/response time must be below 2 seconds:

```
1  oracle type req_res_time {
2    m_max_latency : period;
3    req_res_time(max_latency : period) {
4      m_max_latency = max_latency;
5    }
6    bool evaluate(prr : px_req_res) {
7      return end(px_req_res) - start(px_req_res) <
           m_max_latency;
8    }
9  }
10
11 oracle limit_2s = new req_res_time(2s);
```

C. Results

We made four initial scenarios corresponding to the four fault types described above: an image is not found in storage, an image is not found by one web server (web server 1), a web server becomes slow (web server 2), and a client (client 3) acts maliciously and attempts a denial of service (DoS). The web server slowness is achieved by adding a random delay with an average of $2s$ (the exact limit of the allowed response time) forcing around half of the requests to fail, while allowing around half of the requests to succeed.

In all scenarios, clients 1, 3 and 4 will make requests for a web page, then request all images in it, and then will sleep for a random amount of time taken, respectively, from the distributions $N(2,0.5)$, $N(1.75,0.5)$ and $N(1.75,0.5)$. This usage reflects a somewhat faster pace than a human would perform (2 seconds between pages) but speeds up convergence of the entropy. If we halve the number of requests, entropy will converge at half the speed but all other factors remain unchanged. In all scenarios except the DoS, client 2 will wait according to $N(2.5,0.02)$. In the DoS, client 2 will wait according to $N(0.2,0.02)$.

We aimed our diagnosis maximum entropy to 0.01 (yielding certainty over 99%). Tables III, IV, V and VI reflect the results of the first four scenarios. They contain the evolution of diagnosis over time with the total number of architecture level computations (dynamic web requests and static web requests) detected, the computed entropy and the main fault candidates. We consider $t = 0$ when the first failure occurs. It can be seen by the experimental data that the algorithm is able to correctly identify the cause of the failure in all scenarios.

Time (s)	Req#	Entropy	Main candidates
1	8	0.823	d1=19.0%, fs=79.7%
2	15	0.704	d1=19.0%, fs=81.0%
3	20	0.036	fs=99.7%
4	29	0.067	fs=99.4%
5	31	0.054	fs=99.5%
6	36	0.064	fs=99.4%
7	44	0.006	fs=100%

TABLE III. RESULTS OF SCENARIO 1: FAULTY FILE SYSTEM.

Time (s)	Req#	Entropy	Main candidates
1	14	1.211	web1=63.9%, d1=29.6%
2	18	1.226	web1=68.5%, d1=18.3%, fs=13.1%
3	20	1.078	web1=74.9%, d1=10.6%, fs=14.3%
4	29	0.647	web1=87.5%, d1=3.3%, fs=9.1%
(rows omitted for brevity.)			
18	110	0.058	web1=99.4%, d1=0.4%, fs=0.2%
19	119	0.019	web1=99.8%
20	124	0.008	web1=99.9%

TABLE IV. RESULTS OF SCENARIO 2: FAULTY WEB SERVER 1.

Tables VII and VIII contain the results for the fifth scenario with and without correlation fault detection, respectively. In the first case we can see that, because only the web server *or* the dispatcher (or both but independently) can be responsible, the system is not able to attain very low entropy values. It blames with higher probability the web server because the load balancer participates in more successful computations. If we enable correlated fault detection then we can accurately determine that the fault happens when *both* the dispatcher and web server are used together.

The evaluation of the scenario results allow us to draw three main conclusions:

- Is possible to recognize high-level architectural transactions from lower-level events using our recognition language;
- Entropy computation provides a good way to detect when enough information has been collected for diagnosis;
- Correlated faults can be detected albeit at expense of some diagnosis time penalty.

The existence of diagnosis time penalty is theoretically predictable: because component combinations increase significantly, the number of candidates considered for fault localization increases and, consequently, more data is required.

Time (s)	Req#	Entropy	Main candidates
1	5	1.525	web2=63.5%, d1=12.2%, c4=12.2%, fs=12.2%
2	10	1.837	web2=42.3%, c4=24.1%, d1=24.1%, fs=1.0%
3	12	1.912	web2=35.9%, c4=25.9%, d1=25.9%, fs=12.3%
4	16	1.799	web2=43.3%, c4=31.3%, d1=14.8%, fs=10.6%
5	18	1.886	web2=34.5%, c4=34.5%, d1=18.5%, fs=12.5%
6	18	1.885	web2=34.5%, c4=34.5%, d1=18.5%, fs=12.6%
7	25	0.831	web2=76.6%, d1=22.9%
(rows omitted for brevity.)			
21	63	0.671	web2=84.1%, d1=15.5%
22	70	0.0285	web2=99.8%
23	71	0.0266	web2=0.99.8%
24	75	0.0257	web2=99.8%
25	79	0.006	web2=100%

TABLE V. RESULTS OF SCENARIO 3: WEB SERVER 2 IS SLOW.

Time (s)	Req#	Entropy	Main candidates
1	25	0.000	c2=100%

TABLE VI. RESULTS OF SCENARIO 4: CLIENT 2 TRIES A DOS.

Time (s)	Req#	Entropy	Main candidates
1	7	1.889	web1=40.7%; fs=23.1%; c3=23.1%; d1=13.1%
2	10	1.750	web1=52.4%; fs=16.9%; d1=16.9%; c3=13.9%
3	15	1.253	web1=65.7%; d1=22.6%; fs=11.5%
4	19	1.103	web1=73.6%; d1=14.5%; fs=11.8%
5	21	1.059	web1=75.1%; d1=14.8%; fs=9.9%
6	24	0.874	web1=81.8%; fs=9.1%; d1=9.1%
7	30	0.847	web1=82.6%; d1=9.2%; fs=8.1%
8	34	0.711	web1=86.5%; fs=6.7%; d1=6.7%
9	38	0.808	web1=83.7%; d1=9.3%; fs=6.9%
10	38	0.808	web1=83.7%; d1=9.3%; fs=6.9%
11	41	0.925	web1=80.1%; d1=10.9%; fs=8.9%
12	41	0.925	web1=80.1%; d1=10.9%; fs=8.9%
13	52	0.120	web1=98.4%; d1=1.6%
14	52	0.120	web1=98.4%; d1=1.6%
(rows omitted for brevity.)			
28	115	0.132	web1=98.2%; d1=1.8%
29	120	0.126	web1=98.3%; d1=1.7%
30	123	0.145	web1=97.9%; d1=2.1%
31	127	0.130	web1=98.2%; d1=1.8%
(rows omitted for brevity.)			
54	212	0.168	web1=97.5%; d1=2.5%
55	221	0.133	web1=98.2%; d1=1.8%
56	222	0.129	web1=98.2%; d1=1.8%

TABLE VII. RESULTS OF SCENARIO 5 WITHOUT CORRELATION DETECTION: WEB SERVER 1 IS SLOW WHEN REQUESTS COME FROM DISPATCHER 1.

Time (s)	Req#	Entropy	Main candidates
1	3	3.168	db+c4=12.7%; db+d1=12.7%; db+web1=12.7%; db=12.7%; c4+d1=12.7%; c4+web1=12.7%; c4=12.7%
2	8	2.894	db+c4=19.3%; c4+d1=19.3%; c4+web1=19.3%; c4=19.3%; db+d1=6.5%; db+web1=3.7%; db=3.7%
3	11	3.127	db+c4=20.0%; c4+web1=20.0%; c4+d1=11.4%; c4=11.4%; db+d1=8.2%; db+web1=6.4%; db=6.4%
4	15	3.089	db+c4=20.6%; c4+web1=20.6%; c4+d1=11.7%; c4=11.7%; db+d1=8.5%; db+web1=6.6%; db=5.4%
5	20	3.011	db+c4=22.0%; c4+web1=22.0%; c4+d1=12.5%; c4=12.5%; db+web1=7.1%; db+d1=5.8%; db=4.3%
6	24	1.902	c4+web1=48.8%; c4+d1=21.4%; c4=21.4%
7	27	1.430	c4+web1=73.0%; c4+d1=7.7%; c4=7.7%
8	31	2.119	d1+web1=63.2%; web1=14.8%
9	31	2.119	d1+web1=63.2%; web1=14.8%
(rows omitted for brevity.)			
28	97	0.905	d1+web1=80.4%; web1=17.0%
29	100	0.818	d1+web1=83.8%; web1=13.5%
30	104	0.865	d1+web1=82.1%; web1=15.2%
(rows omitted for brevity.)			
57	220	0.095	d1+web1=98.9%; web1=0.9%
58	223	0.094	d1+web1=99.0%; web1=0.9%
59	223	0.094	d1+web1=99.0%; web1=0.9%
60	227	0.103	d1+web1=98.8%; web1=1.0%
(rows omitted for brevity.)			
85	328	0.024	d1+web1=99.8%; web1=0.2%
86	332	0.012	d1+web1=99.9%; web1=0.1%
87	337	0.012	d1+web1=99.9%; web1=0.1%
88	343	0.012	d1+web1=99.9%; web1=0.1%
89	344	0.012	d1+web1=99.9%; web1=0.1%
90	350	0.010	d1+web1=99.9%
91	356	0.009	d1+web1=99.9%

TABLE VIII. RESULTS OF SCENARIO 5 WITH CORRELATION DETECTION: WEB SERVER 1 IS SLOW WHEN REQUESTS COME FROM DISPATCHER 1.

The fault localization output will, as expected, converge more slowly. However, even in this scenario, a detection with 99.9% certainty in 90*s* of a web server which slows half of the requests only when used with a certain dispatcher is still an encouraging result.

V. RELATED WORK

Diagnosis in software systems is currently addressed both directly and indirectly. It is addressed directly in hand-crafted techniques usually aimed at improving quality attributes in systems and in design-time techniques that aim at identifying faults in developed code. Diagnosis is also addressed indirectly in the general field of self-adaptive systems as part of repair-based techniques.

A typical approach to diagnosis in software systems is to develop special-purpose diagnostic mechanisms for a particular class of system and particular classes of faults. For example, the Google File System [13] and Hadoop [8] use fast, local recovery and replication to achieve high availability for scalable distributed file systems for data-intensive applications. These systems use custom-built monitoring and diagnosis to determine failures of individual servers. While such hand-crafted techniques are typically very effective for the specific kind of system they address, (1) they do not generalize to other systems where the same architectural assumptions do not hold, and (2) they usually assume single-fault scenarios.

Other approaches use simple heuristics to perform diagnosis. Both software rejuvenation [17], [25] and recovery-oriented computing [5] fall in this category. Software rejuvenation selectively restarts components when certain measurements, for example, memory usage, degrades. Recovery-oriented computation uses statistical machine learning techniques to perform diagnosis. These techniques have the advantage of being easy to calculate and are often widely applicable, but they lack precision, resulting in inefficiencies and poor coverage.

An indirect approach to diagnosis is done by repair handlers in self-adaptive systems. For example, the Rainbow system incorporates a set of repair strategies that are triggered when certain architectural invariants are violated in a running system [7], [12]. Each strategy is responsible for determining whether to correct the problem at hand, and if so, how. In order to do this a strategy has to carry out its own fault diagnosis and localization. But this has the disadvantage that each repair handler must do its own diagnosis, possibly adding to run-time overhead (if multiple strategies are used), greatly increasing the effort required to produce repair handlers, and relying on the strategy writer to get the diagnosis right. Similarly, in the three-layer architecture model proposed in [19] higher level planning mechanisms are responsible for diagnosis once a problem has been detected.

None of these techniques provides a general, systematic basis for *run-time* fault diagnosis. In contrast, there has been considerable research on automatic fault diagnosis used at *design time*. Traditionally, automatic approaches to software fault localization are based on using a set of observations collected during the testing phase of system development to yield a list of likely fault locations, which are subsequently used by the developer to focus the debugging process [23]. Existing approaches can be generally classified as either statistics-based or model-based. The former uses an abstraction of program traces, collected for each execution of the system, to produce a list of fault candidates [20], [16], [21]. The latter combines a *model* of the expected behavior with a set of observations to compute a diagnostic report [11], [22].

Model-based approaches are more accurate than statistical ones, but are much more computationally demanding (in both time and space), and they require detailed models of the correct behavior of the system under test. Recently a novel reasoning technique over abstractions of program traces, combining the best characteristics of both worlds, has been proposed [4].

It has low time/space complexity (like statistics-based techniques), yet with high diagnostic accuracy (like reasoning techniques). As we have described, such properties make the technique especially amenable to (continuous) run-time analysis. In this paper, we refer to this kind of reasoning technique as spectrum-based multiple fault localization (SMFL), which we use as the basis for our diagnosis.

VI. Conclusions

In this paper we have described an approach for autonomic diagnosis of faults in a system. We developed a language that allows system behavior to be described and which can be composed hierarchically, facilitating reuse among systems. This language, based on first-order logic, is very expressive and allows the definition of a very large set of system behaviors.

We have also provided an algorithm that automatically adjusts the amount of data required for diagnosis, the window of observation of the system, to attain a predefined level of certainty specified as a maximum value of entropy of fault candidates. We further provided a technique that extends the existing SMFL algorithm to detect correlated faults between components in a system.

Our research on autonomic diagnosis of run-time failures raises several questions that will need to be addressed in future work. Our correlated fault detection algorithm is, theoretically, exponential in the number of components if all possible correlations are to be found. However, use of architecture structure may restrict this significantly and this is worth further investigation.

Also, the target of diagnosis itself can be further improved. We do not have support for hierarchical structures (components and connectors inside components and connectors), but many system definitions take advantage of hierarchical decomposition to improve ease of understanding and reasoning. We are also currently targeting only architecture elements in the dynamic perspective but this work can be extended to support both static elements (such code / libraries) and physical elements (such as servers and network).

Finally, we plan to study how our behavior recognition language can be integrated with the concept of *architectural styles*. Architectural styles are one of the basic foundations for reuse in software architecture and the work we developed shows several visible connections that should be explored in future work.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. CNS 1116848.

References

[1] R. Abreu and A. J. C. van Gemund. A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In V. Bulitko and J. C. Beck, editors, *Proceedings of the 8th Symposium on Abstraction, Reformulation and Approximation (SARA'09)*, Lake Arrowhead, California, USA, 8 – 10 July 2009. AAAI Press.

[2] R. Abreu and A. J. C. van Gemund. Diagnosing multiple intermittent failures using maximum likelihood estimation. *Artif. Intell.*, 174(18):1481–1497, 2010.

[3] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. An observation-based model for fault localization. In *Proc. of WODA'08*. ACM Press, July 2008.

[4] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. Spectrum-based multiple fault localization. In G. Taentzer and M. Heimdahl, editors, *Proc. of ASE'09*. IEEE Computer Society, 2009.

[5] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot: A technique for cheap recovery. In *Proc. OSDI'04*, San Francisco, CA, 2004.

[6] P. Casanova, B. R. Schmerl, D. Garlan, and R. Abreu. Architecture-based run-time fault diagnosis. In I. Crnkovic, V. Gruhn, and M. Book, editors, *ECSA*, volume 6903 of *Lecture Notes in Computer Science*, pages 261–277. Springer, 2011.

[7] S.-W. Cheng, D. Garlan, and B. Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. In *Proc. of SEAMS'06*, 21-22 May 2006.

[8] D. Cutting. The hadoop framework, 2010.

[9] J. de Kleer. Diagnosing intermittent faults. In G. Biswas, X. Koutsoukos, and S. Abdelwahed, editors, *Proceedings of the 18th International Workshop on Principles of Diagnosis (DX'07)*, pages 45 – 51, Nashville, Tennessee, USA, 29 – 31 May 2007.

[10] J. de Kleer. Diagnosing multiple persistent and intermittent faults. In *Proc. of IJCAI'09*. AAAI Press, July 2009.

[11] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[12] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self adaptation with reusable infrastructure. *IEEE Computer*, 37(10), October 2004.

[13] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the Symposium on Operating Systems Principles*, New York, NY, USA, 2003. ACM.

[14] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10(3):171–194, 2000.

[15] IBM. An architectural blueprint for autonomic computing. White paper, 2006.

[16] J. A. Jones, M. J. Harrold, and J. T. Stasko. Visualization of test information to assist fault localization. In *Proc. of ICSE'02*. ACM Press, May 2002.

[17] N. Kolettis and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *Proc. of FTCS'95*, Washington, DC, USA, 1995. IEEE Computer Society.

[18] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29:155–163, 1988.

[19] J. Kramer and J. Magee. A rigorous architectural approach to adaptive software engineering. *J. Comput. Sci. Technol.*, 24:183–188, March 2009.

[20] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proc. of PLDI'05*, Chicago, Illinois, USA, 2005.

[21] C. Liu, L. Fei, X. Yan, J. Han, and S. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering (TSE)*, 32(10):831–848, 2006.

[22] W. Mayer and M. Stumptner. Evaluating models for model-based debugging. In *Proc. of ASE'08*, 2008.

[23] M. Palviainen, A. Evesti, and E. Ovaska. The reliability estimation, prediction and measuring of component-based software. *Journal of Systems and Software*, 84(6):1054 – 1070, 2011.

[24] C. E. Shannon and W. Weaver. *The Mathematical Theory of Communication*. Univ of Illinois Press, 1949.

[25] K. S. Trivedi and K. Vaidyanathan. Software aging and rejuvenation. In *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons, Inc., 2008.