9-2011

Architecture-Based Run-Time Fault Diagnosis

Paulo Casanova
Carnegie Mellon University

Bradley Schmerl
Carnegie Mellon University

David Garlan
Carnegie Mellon University

Rui Abreu
Universidade do Porto

Published In
Lecture Notes in Computer Science, 6903, 261-277.

In Proceedings of the 5th European Conference on Software Architecture, 13-16 September 2011.

Architecture-based Run-time Fault Diagnosis

Paulo Casanova[1], Bradley Schmerl[1], David Garlan[1], and Rui Abreu[2]

[1] School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{paulo.casanova|schmerl|garlan}@cs.cmu.edu
[2] Department of Informatics Engineering
Faculty of Engineering of University of Porto
Porto, Portugal
rui@computer.org

Abstract. An important step in achieving robustness to run-time faults is the ability to detect and repair problems when they arise in a running system. Effective fault detection and repair could be greatly enhanced by run-time fault diagnosis and localization, since it would allow the repair mechanisms to focus adaptation effort on the parts most in need of attention. In this paper we describe an approach to run-time fault diagnosis that combines architectural models with spectrum-based reasoning for multiple fault localization. Spectrum-based reasoning is a lightweight technique that takes a form of trace abstraction and produces a list (ordered by probability) of likely fault candidates. We show how this technique can be combined with architectural models to support run-time diagnosis that can (a) scale to modern distributed software systems; (b) accommodate the use of black-box components and proprietary infrastructure for which one has neither a specification nor source code; and (c) handle inherent uncertainty about the probable cause of a problem even in the face of transient faults and faults that arise only when certain combinations of system components interact.

Keywords: Autonomic computing, diagnosis, software architecture, run-time.

1 Introduction

With increasing reliance on software-based systems to support virtually all aspects of our daily lives, an important new requirement for these systems is the ability to detect and resolve problems at run time. This requirement has spawned an active area of research in autonomic computing [19, 6, 12].

Autonomic computing is based on the idea of turning ordinary software systems into closed-loop control systems. That is, systems are monitored to provide observations of their run-time behavior. Those observations are then analyzed at run-time in reference to models of desired or expected behavior. If significant deviations are observed, repair actions are executed to correct the problems.

When designing an autonomic system, a key issue is the kinds of models that are used in the control layer at run time to capture observed system behavior and detect problems. Research carried out over the past decade has demonstrated that software

In Proceedings of the 5th European Conference on Software Architecture, 13-16 September 2011.

architectures can be particularly effective in this capacity [14, 16, 27].Architectures provide a high-level view of the system, reducing the complexity of understanding what the system is doing, and supporting scalability to complex distributed systems. Suitably annotated architectures also permit the autonomic decision-making apparatus to detect the presence of systemic problems and trends, such as degraded performance. Finally, architectures allow one to capture common patterns of repair that are tuned to the specific style of system and its implementation.

While such "architecture-based self-adaptation" shows great potential, one outstanding problem is *diagnosis:* determining the likely causes of a detected problem. By narrowing the scope of concern for repair to candidates that are the probable cause of an observed problem, the ability to effectively adapt to a problem can be greatly increased.

Run-time diagnosis for architecture-based self-repair, however, is particularly challenging. First, the presence of concurrency makes it difficult to identify which of many possible computations might have caused a problem. Second, reliance on middleware for distributed communication, and more generally the use of components and infrastructure produced by many organizations, means that in many cases neither specifications nor code is available for all parts of the system. Third, in many systems, problems may be intermittent, caused by transient faults or variability in loads. Moreover, while some faults may be directly traceable to a single component (such as a crashed server), in general the source of a problem may be a result of certain combinations of elements (e.g., a specific server interacting with a specific database).

In this paper we describe a systematic approach that adapts a reasoning technique called *spectrum-based multiple fault localization* (SMFL) to architecture-based self-repair. SMFL is a lightweight technique that takes as its input a form of trace abstraction and produces a list of likely fault candidates, ordered by probability of being the true fault explanation [1, 4]. Impressive diagnostic results for *design-time* testing and debugging of both hardware and software systems have been achieved using SMFL [1]. However, there has been little work in applying these results in the context of *run-time* detection and repair, especially in the context of architecture-based adaptation. As we describe in the remainder of this paper, key features of our approach include: (a) the ability to define at a high level what kinds of behavior to use as the basis for fault localization; (b) the ability to associate such behaviors with families of systems (or architectural styles), allowing reuse of the specifications for all instances of the family; and (c) the ability to take into consideration quality attributes, such as performance and availability, in determining both the presence and cause of a problem.

2 Related Work

One of three approaches to fault diagnosis has been typically adopted by autonomic systems. One is to use simple heuristics. For example, software rejuvenation [21, 30] is a technique where components are selectively restarted, using the heuristic of choosing the longest running (i.e., oldest) components to reboot next. This technique can generally improve the robustness of a system, since in many cases faults occur because parts of a system may degrade over time (due, for example, to memory leaks). However, clearly not all faults in a system are a result of aging. Thus heuristics have the advantage of

In Proceedings of the 5th European Conference on Software Architecture, 13-16 September 2011.

being easy to calculate and often widely applicable, but they lack precision, resulting in inefficiencies and poor coverage.

A second approach is to develop special-purpose diagnostic mechanisms for a particular class of system and particular classes of faults. For example, recovery-oriented computing [5] uses a form of local rebooting that takes advantage of the particular characteristics of JEE-based systems, where built-in persistence mechanisms allow computations to be terminated and restarted without loss of data. Diagnosis in these systems uses statistical machine learning techniques to identify a specific component to restart – again taking advantage of the specific features of JEE systems. Similarly, the Google File System [15] 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 assume single-fault scenarios.

A third approach allocates the task of diagnosis to individual repair handlers. 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, 14]. 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. For example, a strategy triggered by high latencies might attempt to reboot faulty servers. But before it can do that it needs to figure out which servers (if any) might be failing. Associating diagnostics with the repair mechanism has the advantage that diagnosis can be specialized to the needs of the particular kind of repair. But it 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 [23] 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 *development 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 [28]. 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 [24, 18, 25]. The latter combines a *model* of the expected behavior with a set of observations to compute a diagnostic report [11, 26].

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 will see, 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). Previous research efforts into SMFL have focused primarily on helping developers fix bugs at development time, where one can easily identify the start and end of a given test case, as well as which elements were involved in the execution of the test case. To our knowledge, none of these have been combined with architecture models to support run-time diagnosis, where, as opposed to development time, detecting a given execution is difficult (e.g., due to concurrency).

3 Approach

Applying SMFL at run time to diagnose problems relative to architectural models raises a number of challenges. First, we need to be able to identify the beginning and end of computations in the system that we are interested in. This is challenging because interactions may be interleaved and concurrent. Second, we need to be able to relate these run-time interactions, which are in terms of system level events, with their corresponding elements in an architecture model. In this section, we give a brief overview of our approach; in later sections we elaborate on the details.

To illustrate the ideas, consider a family of systems, whose architecture is illustrated in Figure 1, in which a variable number of clients can interact with a pool of servers that have access to a common data store. Client HTTP requests are mediated by one or more dispatchers, selected randomly by a client, which forward requests to a specific server in the pool. Although relatively simple in structure, such systems are representative of a large class of applications, and illustrate some of the challenges for run-time diagnosis. First, such a system could have hundreds of clients and servers (for example, running on a cloud computing platform), handling thousands of simultaneous requests. This makes it challenging to determine which elements are involved with a particular request. Second, when problems occur, it is important to pinpoint the causes quickly, since a problem with a dispatcher (for example), could drastically impact the overall ability of the system to deliver its services in a timely manner. Third, there are many sources of uncertainty inherent in this system. For example, high latency in handling customer requests could be caused by faults, or combinations of faults, in any number of components. Fourth, although certain kinds of problems may be easily detected and fixed (such as a server crash), softer intermittent failures causing high latencies are equally as important to detect and repair.

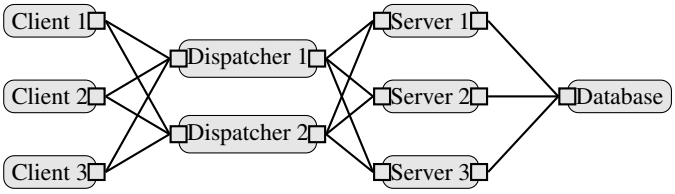

Fig. 1. Simplified Web Server Example.

Our approach to adapting SMFL to support architecture-based fault detection and localization uses the following three steps:

In Proceedings of the 5th European Conference on Software Architecture, 13-16 September 2011.

1. First, we define a collection of *transaction types* for the style of system under analysis using parametric architecture behavior descriptions. Each transaction type specifies (a) a set of finite computational paths through the architecture, and (b) the criteria for determining whether a given computation of that type has succeeded or failed. For the class of system shown in Figure 1, one possible transaction type would represent the normal client-server response sequence: a client initiates a request, which is handled by a dispatcher, dispatched to a particular server, and then returned back to the client. A success criterion might be that the client should receive a reply within a certain number of seconds.
2. Next we provide a way to monitor the running system from an architectural perspective, adapting prior work on architecture-based monitoring and fault detection [29]. This involves using probes and event monitoring mechanisms in the running system to determine (a) when a complete transaction has occurred, (b) the architectural elements involved, and (c) whether the transaction succeeded. For the example we would record the specific client, dispatcher, and server involved in the transaction, and whether the latency threshold was exceeded.
3. The results of spectrum monitoring are then accumulated in a fault localization phase. Adapting SMFL algorithms for the run-time setting, probabilistic rankings of likely fault causes (if any) are calculated. These can then be used to trigger repair mechanisms. In the example, the fault localization algorithms might determine, for example, that with probability 0.8 the cause of an intermittent latency problem is the combination of dispatcher 2 interacting with server 5.

To elaborate on this approach, in the following sections we first summarize the key ideas behind SMFL. (For a detailed description see [4].) Then we explain in more detail how we carry out these three parts of our approach.

4 Spectrum-based Reasoning for Fault Localization

Fault localization based on reasoning over program spectra is characterized by the use of (a) *program spectra*, abstracting from actual observation variables, structure, and component behavior; (b) a low-cost, heuristic reasoning algorithm, STACCATO [4] to extract the significant set of *multiple-fault candidates*; and (c) abstract, intermittent models, that take into account that a faulty component may behave correctly with a specific probability, to compute the *candidate probability* of being the true fault.

4.1 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 [22]). Although we work with these so-called component-hit spectra, the approach outlined in this section easily generalizes to other types of program spectra [17].

Program spectra are collected for N (pass/fail) executions of the system. Both spectra and program pass/fail information are input to spectrum-based fault localization. The program spectra are expressed in terms of a $N \times M$ *activity matrix* A, for example in Table 1. 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. The pass/fail information is stored in a vector e, the *error vector*, where e_i signifies whether run i has *passed* ($e_i = 0$) or *failed* ($e_i = 1$). Note that the pair (A, e) is the only input to the spectrum-based diagnosis approach.

4.2 Candidate Generation

As in 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. The above *weak* model[3] 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 1. 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 1 we obtain the following health propositions for h_j:

[3] 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 Proceedings of the 5th European Conference on Software Architecture, 13-16 September 2011.

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

These health propositions have a direct correspondence with the original matrix structure. 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, aka 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 [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].

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.

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

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

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.)

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

$$\Pr(obs|d_1) = (1 - h_1 \cdot h_2) \cdot (1 - h_2) \cdot (1 - h_1) \cdot h_1 \tag{5}$$

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

In Proceedings of the 5th European Conference on Software Architecture, 13-16 September 2011.

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 $D =< \{1,2\},\{1,3\} >$.

5 Adapting SMFL to Architecture-based Run-time Diagnosis

On the surface of it, combining SMFL with architecture-based adaptation would appear to be a natural synthesis. Architecture models, on the one hand, provide an abstract component-oriented view that can form the basis for a scalable representation of the elements that might contribute to faulty behavior. Further, architecture-based monitoring and fault detection (but not diagnosis) are reasonably well established [29]. SMFL, on the other hand, provides a light-weight, efficient, statistical approach that supports diagnosis in the face of uncertainty, coordinated faults, and transient errors. Further, SMFL is agnostic about the nature of a fault, allowing systemic properties based on quality attributes (such as performance) to guide the ranking procedure.

However, there are a number of obstacles that must be overcome to synthesize these two disciplines. First, there needs to be some way to *define the traces of interest*: one must be able to describe what kinds of computations should be monitored, as well as the criteria for determining whether a computation has succeeded or failed. Moreover, while SMFL expects *finite* traces, in general the behavior of a running system is not finite (or so one hopes). Second, there must be a way to *detect the occurrence of traces in the running system*. As noted earlier, concurrency makes this difficult, since many simultaneously executing traces may be present in a system. (Recall that in the development time context for which SMFL was originally created, each trace can be observed as a separate run of the system under test.) Third, the algorithm for performing SMFL must be adapted to handle concurrently executing traces, and provide an appropriate window of observation (as described later).

5.1 Defining Transactions

Recall that SMFL expects as input a series of spectra, where each spectrum is a finite set of components that participated in a given computation (a finite program trace), together with an indication of its status (pass/fail). How can we define such computations – which will in turn serve as the basis for monitoring, diagnosis, and fault localization?

The problem is non-trivial for two reasons. First, a trace defines a *finite* execution. However, we are interested in systems that operate continuously, so that at a system level the behavior of the system is infinite. Second, different kinds of systems embody very different kinds of computational models. For example, complex computations in a service-oriented architecture (SOA) are often defined by an orchestration script, which indicates how the various components are coordinated, and how data passes from one to another. In contrast, a system based on sensor networks may involve processing streams of sensor readings.

Our approach is based on two key ideas. The first is the idea of a *transaction family*. A transaction family defines a parameterized pattern of behaviors as finite computations, expressed in terms of the architectural elements (components and connectors) that are involved in that computation, and the flow of information/control between them (in a way similar to [20]). An instantiation of that pattern (in terms of specific architectural elements) is an individual transaction. Additionally we associate a set of properties with the components and flows. These properties indicate things like the time that a flow event happened or the load on a server. Finally, a transaction family includes a boolean function that determines whether a given transaction has succeeded.

The second idea is to *associate these transaction families with architectural styles*. An architectural style describes the types of elements and their possible legal associations in a system, which allows architectural patterns referring to those types to be defined. Several transaction families can be defined for each architectural style, each representing a different pattern of computation. Note that the transaction families need not cover *all* of the behaviors of systems in the family – only the ones that are of interest to diagnosis. However, by defining transaction families at the architectural style level, we can immediately reuse diagnosis systems for different systems. And although a different technology may be required to place probes (techniques for probing C programs are different from those for Java programs), both the diagnosis system and its configuration are fully reusable.

There are many possible ways that one might define transaction families, including state machines, process algebras, and so on. In our work we adopt a form of message sequence charts [13]. For example one transaction family for a web-server family that might be used to model the system in Figure 1 would be represented by the sequence diagram in Figure 2. The pattern of communication shown there defines the client-server round-trip execution flow discussed earlier. It involves an arbitrary client, dispatcher, and server, as well as the database. The first three represent parameters of the family which (as we describe below) will be instantiated with specific components during system execution. Properties associated with the family include the time taken to serve a client's request (i.e., the request latency). An associated boolean function returns true if the latency (difference in request and reply times) is under the appropriate threshold.

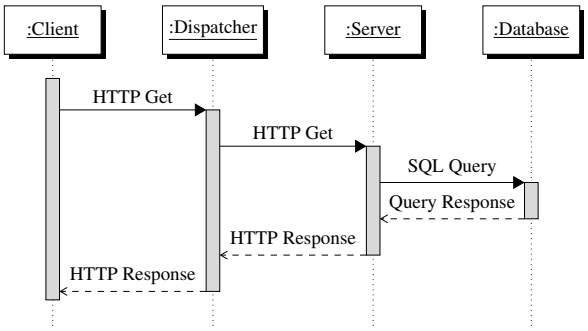

Fig. 2. Example Message Sequence Chart for a HTTP Request Transaction.

Transaction families have several important benefits. First, they involve relatively minimal specification: rather than requiring a full formal account of the architectural

In Proceedings of the 5th European Conference on Software Architecture, 13-16 September 2011.

behavior, we focus only on finite abstract "slices" of it, reducing the overhead of defining relevant behavior and making the approach generally accessible. Second, definition of transaction families for a given system or architectural style can be incremental: it is possible to add new templates or to add detail to an existing template (for example, by including a finer-grained account of the set of elements involved in the transaction). This allows users of the technique to get increased benefits for increased effort. Third, by associating transaction families with architectural families, we amortize the effort of defining behaviors, and allow reuse of prepackaged collections of transaction families for commonly-used architectural styles.

5.2 Detecting Traces

Given a way to specify transactions, we now need a way to observe them in a running system and then use those observations to carry out fault diagnosis. Figure 3 shows the process that we use do to this.

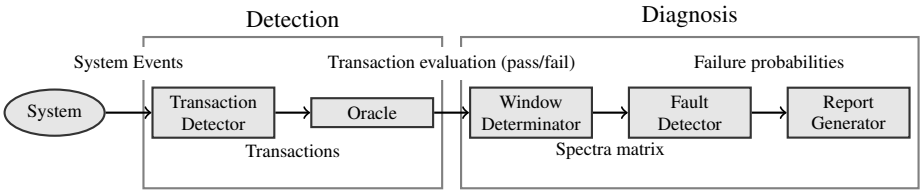

Fig. 3. The Architecture of our Experimental Framework.

To detect transactions, we adapt earlier work in architecture-based monitoring [29]. First, a system is instrumented so that it can be monitored at runtime. Monitored events are placed on an "event bus" where they can be consumed by the detection phase of our diagnostic infrastructure. In the client-server example above, monitored events include activities like initiation of HTTP requests over the client-dispatcher connectors.

To monitor a system, there are numerous mechanisms that can be used that vary in terms of the kind of behavior they detect and the kind of system they are appropriate for. For distributed systems, standard middleware and network communication infrastructure provide mechanisms to monitor communication events and their properties. For systems working on a single host, code-oriented monitoring can be used. For example, aspect-oriented techniques can weave monitoring code into an existing code base (see, for example, [29]). In this research the choice of monitoring mechanism and the placement of relevant probes has not yet been a major focus of our efforts. However, as we discuss later, we view this as an area for future research.

During the detection phase, events are first filtered to extract those relevant to the transaction families that are being observed. Next, events are passed to a detection machine generated from the transaction families. Specifically, adapting earlier work on DiscoTect, we monitor events as a set of concurrent state machines, modeled as Petri Nets [29]. The key idea is that behavior is tracked by moving tokens through a state machine in response to low-level events. When tokens reach certain terminal states the machine emits the set of architectural elements that were involved in the trace and an indication of the transaction type. This information is then fed to an oracle that evaluates the boolean function associated with that spectrum type on the detected spectrum.

5.3 Diagnosis

The second phase of processing is *diagnosis*. This is broken down into three parts. The first part is window determination. This step is responsible for aggregating a sequence of transactions to define the matrices – the (A, e) of Section 4 – that can be analyzed by the SFML algorithms. In determining these matrices it is important to define an appropriate window. If the window is too small, there may be too few transactions for the results to be statistically significant. If the window is too large, it may contain out-of-date transactions that may skew the diagnosis towards past behavior.

There are a number of criteria that might be used to determine this window. In our current experiments we have found that a time-based window works well. That is, we aggregate all spectra within a temporal window. The value for that time bound needs to be determined by experimentation as it is dependent on the rate of system usage: with high transaction rates, a smaller time window can aggregate enough traces, but if the transaction rate is low then we need a larger time window.

Once a window of spectra has been determined, the associated matrices are given to the SMFL algorithm, which calculates a list of candidate fault explanations (if any) ordered by probability of being the likely cause. This is simply a straightforward application of the SFML algorithms described earlier.

Finally the results are passed to a Report Generator, which outputs the results of the SFML analysis: a list with sets of failed components and their associated probabilities. Automated repair mechanisms (or human operators) can then interpret the results in architectural terms.

6 Evaluation

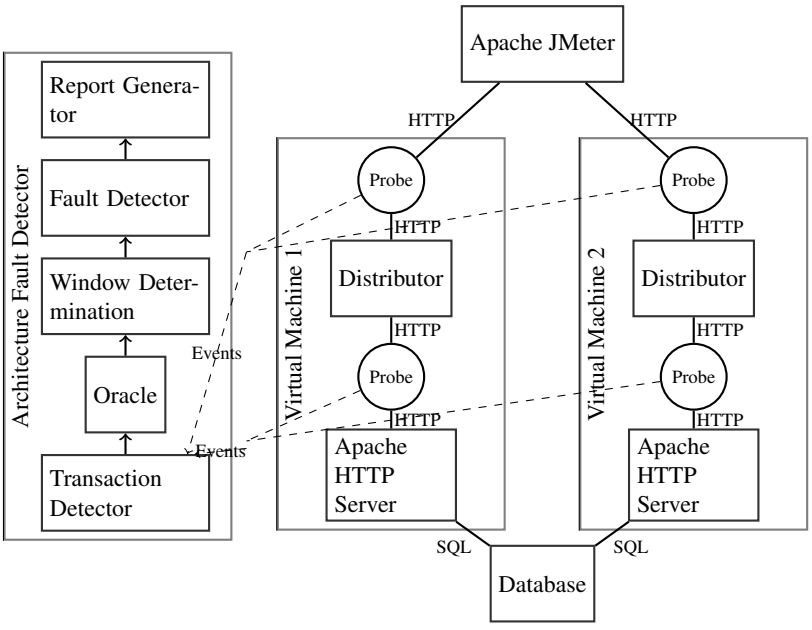

Fig. 4. Evaluation Experiment Setup.

In Proceedings of the 5th European Conference on Software Architecture, 13-16 September 2011.

To evaluate the approach we conducted experiments on a system similar to the example in this paper. In particular, we investigated the hypothesis that the technique could accurately pinpoint problems in a system exhibiting (a) variability in the number of components; (b) distributed system structures that involve realistic, off-the-shelf communication infrastructure and componentry; (c) the presence of transient faults, where failure is based on systemic attributes like end-to-end performance; and (e) faults that might involve more than one component. The combination of these properties yields a system that would be challenging to diagnose given current technology.

One class of problems that fits this criteria are intermittent multi-component faults with additional noise. These problems arise with faulty network connections or application errors that occur only with specific combinations of input data. With these kinds of problems, a fault occurs only sometimes and is generally associated with a specific path on the system. However, other intermittent faults may occur less often due to other reasons, generating additional noise in the spectra. We want to be able to separate out the real errors from the noise.

To create this experiment, we recreated an environment similar to the one Figure 1. In this system, two virtual machines (simulating two servers) run two web servers and dispatchers. The dispatchers choose which web server to send requests to using a round-robin algorithm. An external multi-threaded load test program, *Apache JMeter*, generates requests on both virtual machines simulating clients accessing the system.

A trace family is defined for this system: a *standard* request in which the client performs a request to a dispatcher which forwards it to a web server.

Two interception points, or *probes*, were placed in each machine, one before the request arrives at each dispatcher and one between the dispatchers and the web servers. These interception points (custom-developed based on the *pygmy HTTP server*) add a specific header to the HTTP request to allow tracking the transaction and report to an event bus all events with the component name.

The fault detector receives events from the event bus and uses a Petri Net (PN) to determine to what family the transaction belongs, as previously discussed. The PN used to identify the transaction family is the one in Figure 5. Transitions on the PN are enabled when the corresponding events arrive. A transaction in the PN is initialized with a token in the `START` place and ends when a token arrives at the `DONE:Standard` place. The oracle considers a transaction to be a success if the time elapsed between the request and response is less than 2.5 seconds. This is representative of systems in which response time is a measure of success – systems that do not exhibit easier-to-detect fail-stop failures.

To simulate network delay (or server processing delay), we added a random delay in both *IP1* and *IP2* of the first virtual machine. This means that 25% of all requests receive an added time delay that ensures some of the time they will fail. This generates a hard-to-find problem, namely an intermittent failure on one of the paths: the one containing the *dispatcher 1* (D1) and *web server 1* (WS1). Simultaneously it adds a small (but non-zero) failure probability on both the D1-WS2 and D2-WS1 paths. The experimental results show that the fault detection algorithm is able to statistically separate these results and produce the correct output.

The total number of traces obtained during a run and their distribution between the various components is shown in Table 2(a). As the results show, the D1-WS1 path fails

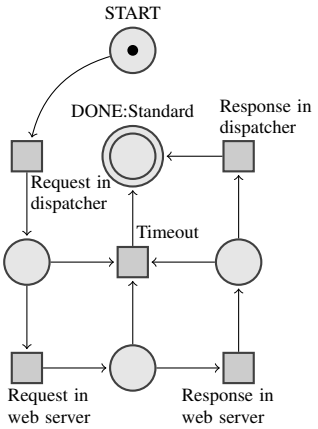

Fig. 5. Petri Net used to identify the transaction family.

26% of the time. The other two paths which include D1 and WS1 fail slightly less than 5% of time time and the D2-WS2 path has no failures.

Table 2. Results.

(a) Number of success/fail spectra for each combination of dispatcher and web server.

	WS1	**WS2**	**Total**
D1	85/31	129/5	**214/36**
D2	117/5	122/0	**239/5**
Total	202/36	251/5	**451/41**

(b) Time evolution of results of failure diagnosis.

Time Window	**Succ./Fail.**	**Diagnosis**
0-10s	30/2	$D1 : 84\%, WS1 : 16\%$
0-20s	119/8	$WS1 : 100\%$
0-30s	201/16	$D1, WS1 : 99\%, D1, D2 : 1\%$

Because SMFL's only input are the spectra, enough data need to be collected before the problem can be detected. In fact, as shown in Table 2(b), the failure probabilities change over time. For example, a 20s window would have determined that WS1 was the only component responsible for the observed failures. Only after 30s is the algorithm able to indict WS1 *and* D1 as the components responsible for the observed failures. This means that window size needs to be carefully chosen so that the SMFL algorithm has enough information to yield accurate diagnosis [2].

7 Conclusions and Future Work

In this paper we described an approach that combines architecture models for monitoring system behavior and spectrum-based fault localization for diagnosing problems. Such a combination provides a systematic, efficient and scalable technique to deal with run-time failures independent of the system domain. Important features of the approach are that it is lightweight, generally applicable to any kind of system, tolerant of uncertainty, and capable of detecting soft anomalies and problems that involve combinations of components.

This line of research raises a number of research questions requiring further investigation. In our current system, probes are manually placed to monitor the activity of

In Proceedings of the 5th European Conference on Software Architecture, 13-16 September 2011.

the running system. We plan to investigate methods for efficient automatic probe placement, including analysis to identify the minimum set of probes required to accurately monitor the spectrum types defined for the system, as well as techniques for dynamic probe placement (e.g., to enable/disable probes at run-time). Our current oracle is determined at design-time, but machine learning-based approaches could provide designs that perform better in adaptive systems. Moreover, as we observed in the experiment, SMFL window size is an important parameter that can affect the accuracy of the diagnosis. We plan to study a systematic, generic method to automatically determine this parameter. We believe that this can be done in a parametric way, based on the family of system and the kind of implementation base on which it is deployed. Furthermore, our architecture-based fault localization approach allows the definition of multiple spectrum types. It is not yet clear whether each type should have its own SMFL diagnosis instance, or whether they should be combined into a single detection component. A key issue will be to determine whether there is a need for multiple SMFL windows depending on spectrum type, as this will require the use of multiple SMFL instances. While the approach scales well in terms of its algorithmic complexity, we plan to conduct experiments on large-scale systems to evaluate the scalability of the method in practice. Finally, we plan to integrate the diagnosis mechanism into detection-diagnosis-repair cycle, to determine how it impacts round-trip self-repair efficiency.

8 Acknowledgements

This research was supported by CyLab at Carnegie Mellon under grants DAAD19-02-1-0389 and W911NF-09-1-0273 from the Army Research Office, and by a grant from the Foundation for Science and Technology via project CMU-PT/ELE/0030/2009 and by FEDER via the "Programa Operacional Factores de Competitividade" of QREN (FCOMP-01-0124-FEDER-012983).

References

1. R. Abreu and A. J. C. van Gemund. Diagnosing multiple intermittent failures using maximum likelihood estimation. *Artif. Intell.*, 174(18):1481–1497, 2010.
2. R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proc. of TAIC PART'07*. IEEE Computer Society, September 2007.
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. B. H. C. Cheng, R. de Lemos, D. Garlan, H. Giese, M. Litoiu, J. Magee, H. A. Müller, M. Pezzè, and R. Taylor, editors. *Proc. of SEAMS 2010*. ACM Press, 2010.
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. S. A. Dobson, J. Strassner, M. Parashar, and O. Shehory, editors. *Proc. of ICAC'09*. ACM Press, 2009.
13. M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2003.
14. 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.
15. 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.
16. D. Ghosh, R. Sharman, H. Raghav Rao, and S. Upadhyaya. Self-healing systems - survey and synthesis. *Decis. Support Syst.*, 42:2164–2185, January 2007.
17. 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.
18. 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.
19. J. Kephart and D. Chess. The vision of autonomic computing. *Computer*, 36(1), 2003.
20. K. Kiviluoma, J. Koskinen, and T. Mikkonen. Run-time monitoring of architecturally significant behaviors using behavioral profiles and aspects. In *Proc. of ISSTA'06*, ISSTA '06. ACM Press, 2006.
21. N. Kolettis and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *Proc. of FTCS'95*, Washington, DC, USA, 1995. IEEE Computer Society.
22. B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29:155–163, 1988.
23. J. Kramer and J. Magee. A rigorous architectural approach to adaptive software engineering. *J. Comput. Sci. Technol.*, 24:183–188, March 2009.
24. 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.
25. 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.
26. W. Mayer and M. Stumptner. Evaluating models for model-based debugging. In *Proc. of ASE'08*, 2008.
27. M. Mikic-Rakic, N. Mehta, and N. Medvidovic. Architectural style requirements for self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, WOSS '02, pages 49–54, New York, NY, USA, 2002. ACM.
28. 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.
29. B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering Architectures from Running Systems. *IEEE Transactions on Software Engineering*, 32(7):454–466, 2006.
30. K. S. Trivedi and K. Vaidyanathan. Software aging and rejuvenation. In *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons, Inc., 2008.