Efficient and Scalable Algorithms for Inferring Likely Invariants in Distributed Systems

Guofei Jiang, Haifeng Chen, and Kenji Yoshihira

Abstract—Distributed systems generate a large amount of monitoring data such as log files to track their operational status. However, it is hard to correlate such monitoring data effectively across distributed systems and along observation time for system management. In previous work, we proposed a concept named flow intensity to measure the intensity with which internal monitoring data reacts to the volume of user requests. We calculated flow intensity measurements from monitoring data and proposed an algorithm to automatically search constant relationships between flow intensities measured at various points across distributed systems. If such relationships hold all the time, we regard them as invariants of the underlying systems. Invariants can be used to characterize complex systems and support various system management tasks. However, the computational complexity of the previous invariant search algorithm is high so that it may not scale well in large systems with thousands of measurements. In this paper, we propose two efficient but approximate algorithms for inferring invariants in large-scale systems. The computational complexity of new randomized algorithms is significantly reduced, and experimental results from a real system are also included to demonstrate the accuracy and efficiency of our new algorithms.

Index Terms—Distributed systems, monitoring data, time series, data management, invariants, randomized algorithms, computational complexity.

1 Introduction

The growing complexity of information systems has significantly increased the difficulty and cost to maintain and manage large-scale distributed systems. According to an IDC study, 20 percent of IT costs were spent on operational system management in 1990 and, now, this percentage is approaching 70 percent [21], [20]. Large information systems such as Google.com and Amazon.com consist of thousands of components including servers, software, networking devices, and storage equipments. Although each of these components is complex enough by itself, the dynamic interaction among them introduces another dimension of complexity. The complexity of information systems originates not only from their scale but also from their dynamics and heterogeneity. For example, user behaviors and loads are always changing, software and hardware components are frequently replaced or upgraded, and a system itself may also include many uncertainties such as caching. Meantime, each information system is integrated with various software and hardware components, which are usually supplied by many different vendors and have their specific configurations. Therefore, it has been a great challenge to maintain and manage distributed systems with such scale and complexity.

During system operation, a large amount of monitoring data can be collected from distributed systems to track their operational status. Software log files, system audit events, and network traffic statistics are typical examples of such monitoring data. If we regard an operational system as a dynamic system like those in control theory [4], this monitoring data can be considered as the observable of its internal states. For example, a fault occurrence could scatter its trace in the monitor data so that we can interpret the monitoring data to observe the fault. Though much monitoring data may have temporal and spatial dependencies, it is hard to correlate such monitoring data effectively across distributed systems and along observation time for system management. One fundamental reason is the difficulty to characterize and model the dynamic behavior of complex systems. In practice, without reasonable models to characterize complex systems, we can hardly import intelligence and develop reasoning capabilities to interpret the monitoring data. We believe that much of this knowledge is inherently system dependent, that is, it is hard to generalize such knowledge across different systems.

In previous work [14], [30], we proposed to model and track transaction flow dynamics in distributed systems for operational system management. Many transaction systems receive millions of transaction requests every day, and these requests flow through system components sequentially according to their application logic. We introduced a concept named *flow intensity* to measure the intensity with which internal monitoring data reacts to the volume of user requests. The number of SQL queries, the number of network packets, and the average CPU usage (per sampling unit) are typical examples of such flow intensity measurements. We calculated flow intensity measurements from monitoring data and proposed an algorithm to automatically search constant relationships between flow intensities measured at various points across distributed systems. If such relationships hold all the time, we regard them as *invariants* of the underlying systems. Invariants can be used to characterize complex systems and support various system management tasks such as

• The authors are with NEC Laboratories America, Inc., 4 Independence Way, Suite 200, Princeton, NJ 08540.
E-mail: {gfj, haifeng, kenji}@nec-labs.com.

Manuscript received 24 Aug. 2006; revised 2 Mar. 2007; accepted 23 July 2007; published online 7 Aug. 2007.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0379-0806.
Digital Object Identifier no. 10.1109/TKDE.2007.190648.

fault detection and isolation (FDI), performance debugging, capacity planning, and resource optimization [13]. Though the invariant search algorithm can run offline to infer invariants from monitoring data, its computational complexity is high so that it may not scale well in large systems with thousands of measurements. In this paper, we propose two efficient but approximate algorithms for inferring invariants in large-scale systems. The computational complexity of the new randomized algorithms is significantly reduced, and experimental results from a real system are also included to demonstrate the accuracy and efficiency of our new algorithms.

2 Related Work

To the best of our knowledge, we believe that we are the first research group to propose the concept of invariant in dynamic systems and its uses for large-scale system management. We have done much work to develop invariant-based frameworks for autonomic system management. We verified that such invariants widely exist in distributed transaction systems such as Internet services, and they can be used to support various system management tasks [13]. In previous work [14], [30], we also gave algorithms and technical details about how such invariants can be used for FDI in complex information systems. Experimental results were also included in these literatures to demonstrate the effectiveness of invariant-based system management solutions. We have been testing such solutions on several large systems and networks including one of the largest systems in the world. In practice, we notice that it is very important to develop computationally efficient algorithms to systematically search invariants in monitoring data, especially for large-scale systems and networks. This motivates the work of this paper.

There is much existing work on the invariant analysis of software programs though the concept of invariants is thoroughly different in this paper. Ernst et al. developed a system named Daikon to discover likely program invariants for supporting program evolution [8], [22]. Daikon infers invariants at specific program points such as procedure entries and exists. It instruments a software program and checks for invariants involving variables in the code. Large test suites have to be generated so as to discover invariants at various program points. Hangal and Lam also developed a tool named DIDUCE to aid programmers in detecting complex program errors and identifying their root causes [10]. DIDUCE instruments a software program and observes its behavior as it runs. It associates with each instrumented program point a set of expressions and dynamically formulates hypotheses of invariants from these expressions. By dynamic invariant detection and checking, DIDUCE is able to detect software bugs and further hunt down the root cause of bugs. However, the concept of invariants and the invariant extraction algorithms are thoroughly different in this paper because we search invariant relationships between flow intensities calculated from the monitoring data in distributed systems rather than invariant expressions in software programs.

In the knowledge and data engineering community, there is much existing work on applying data mining approaches for Web log analysis. Some works use mining techniques to discover Web access patterns and build customer profiles [29], [26], [1]. Some other works mine Web access logs for system performance analysis [28], [25]. Arlitt and Williamson [2] studied workload characterization for Internet Web servers and discovered several workload invariants from six different Web sites. For example, one of such workload invariants is that images and HTML files account for 90–100 percent of the files transferred between clients and Web servers. Menasce et al. [17] defined a hierarchical structure to characterize e-business workloads and examined the statistical and distributional property of workloads from two actual e-business sites. The workload characterization was done at the session, e-business function, and request levels, and they found several invariants in e-business workloads. For example, one of their invariants is that most sessions last less than 1,000 sec. The concept of invariants in this paper is thoroughly different from these workload invariants though we both extract invariants from monitoring data such as log files.

Recently, Jiang et al. [15] introduced a data mining approach called ADMiRe to system performance analysis. They used regression analysis to discover correlations between various system and workload metrics, and the correlation results were then summarized in sets of regression rules. System administrators can manipulate these sets of rules to narrow down problems in performance analysis. Whereas their paper focused on regression rules and algebraic expressions, we use AutoRegressive models with eXogenous inputs (ARXs) [16] to systematically search invariant relationships between flow intensities measured at various points across large-scale distributed systems. They only considered spatial correlations between system metrics in their regression analysis. Instead, ARX models in this paper can characterize temporal dependencies, as well as spatial dependencies, between measurements. This is important because there exist time delays in correlating various measurements across distributed systems, and various system components also may not have a standard clock. We also introduce a sequential testing process to validate the robustness of invariants along time. Besides that, in this paper, we focus on the scalability and efficiency issues of invariant search algorithms, which were not addressed in their work.

3 Flow Intensities and Invariants

Many large transaction systems receive millions of transaction requests every day. The large volume of user requests flow through system components sequentially according to the application software logic. If we regard the control flow graph of application software as a pipe network, the mass of user requests that flows through various software paths can be considered as fluid flowing through that pipe network. During system operation, various system components produce a large amount of monitoring data such as log files to track their operational status. Our first observation is that much of internal monitoring data reacts to the volume of user requests accordingly. For example, network traffic volume and CPU usage usually go up and down in accordance with the volume of user requests. Here, we use flow intensity to measure the intensity with which internal monitoring data reacts to the volume of user requests. Flow intensities can be calculated from the monitoring data

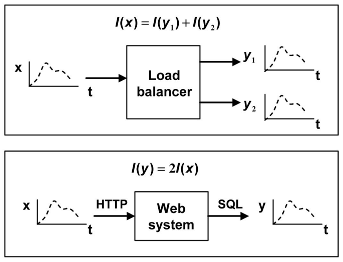


Fig. 1. Correlation of flow intensities.

collected at various points across distributed systems. The number of HTTP requests, the number of SQL queries, and the number of network packets (per sampling unit) are typical examples of such flow intensity measurements. In our experiments, we collect as many as over 100 flow intensity measurements from a typical three-tier Web system. More flow intensity examples are included in Section 9.

Our second observation is that there exist strong correlation or cointegration between many of the flow intensity measurements. This is because these measurements mainly respond to the same external factor—the volume of user requests. The correlation between the flow intensities measured at the input and output of a component could well reflect the constraints the component should bear. As an engineered system, a distributed system imposes many constraints on the relationships among flow intensity measurements. Such constraints could result from many factors such as hardware capacity, application software logic, system architecture, and functionality. Two examples are shown in Fig. 1 to illustrate such constraints. Note that here, we use $I(x)$ to represent the flow intensity measured at point x. Assuming that a load balancer has one input x and two outputs y_1 and y_2, we should always have $I(x) = I(y_1) + I(y_2)$ because this is the physical property of a load balancer. Meantime, in a Web system, if a specific HTTP request x always leads to two related SQL queries y, we should always have $I(y) = 2I(x)$ because this logic is written in its application software. No matter how these flow intensities change in accordance with varying user loads, such relationships (the equations) of the flow intensities are always constant. Our previous work [13] verified that such invariant relationships widely exist in real distributed systems, which are governed by the physical properties or logic constraints of system components.

We model and search constant relationships between flow intensities measured at various points across distributed systems. If the modeled relationships hold all the time, they are regarded as invariants of the underlying systems. In the above examples, the relationships (equations) $I(x) = I(y_1) + I(y_2)$ and $I(y) = 2I(x)$ but not the flow intensities are considered as invariants. Note that unlike a flow intensity itself, the validity of an invariant is not affected by varying user loads. In order to support operational system management, we have to understand basic properties of target systems. However, as discussed earlier, it is extremely hard to characterize large, dynamic, and complex systems in a holistic way. Rather than modeling the whole system, we characterize many local properties of system components. Each invariant is able to capture some local properties of its related components. Therefore, if we can discover a large number of invariants from distributed systems, the combination of these invariants enables us to characterize a complex system well. Further, we can interpret system operational status in real time by tracking the changes of these invariants.

4 Models of Invariants

To make this paper self-contained, we will introduce the models of invariants and discuss our previous invariant extracting algorithm in the next two sections before we propose our new algorithms. For convenience, in the following sections, variables such as x and y are used to represent flow intensity measurements, and we use equations such as $y = f(x)$ to represent invariants. With flow intensities measured at various points across systems, we need to consider how to model the correlation between these measurements. As mentioned earlier, many of such measurements change in accordance with the volume of user requests. As a time series, these measurements should have similar evolving curves with the workload curve along time t. Therefore, we believe that many of the flow intensities should have linear relationships. In this paper, we use ARXs to learn linear relationships between flow intensity measurements.

At time t, we denote the flow intensities measured at the input and output of a component by $x(t)$ and $y(t)$, respectively. The ARX model describes the following relationship between two flow intensities:

$$\begin{aligned} & y(t) + a_1 y(t-1) + \cdots + a_n y(t-n) \\ = \; & b_0 x(t-k) + \cdots + b_m x(t-k-m), \end{aligned} \tag{1}$$

where $[n, m, k]$ is the order of the model, and it determines how many previous steps are affecting the current output. a_i and b_j are the coefficient parameters that reflect how strongly a previous step is affecting the current output. Let us denote

$$\theta = [a_1, \cdots, a_n, b_0, \cdots, b_m]^T, \tag{2}$$

$$\begin{aligned} \varphi(t) = \; & [-y(t-1), \cdots, -y(t-n), \\ & x(t-k), \cdots, x(t-k-m)]^T. \end{aligned} \tag{3}$$

Then, (1) can be rewritten as

$$y(t) = \varphi(t)^T \theta, \tag{4}$$

where T refers to matrix transposition.

Assuming that we have observed two flow intensity measurements over a time interval $1 \leq t \leq N$, let us denote this observation by

$$O_N = \{x(1), y(1), \cdots, x(N), y(N)\}. \tag{5}$$

For a given θ, we can use the observed inputs $x(t)$ to calculate simulated outputs $\hat{y}(t|\theta)$ according to (1). Thus, we can compare the simulated outputs with the real observed outputs and further define the estimation error by:

$$\begin{aligned} E_N(\theta, O_N) &= \frac{1}{N}\sum_{t=1}^{N}(y(t) - \hat{y}(t|\theta))^2 \\ &= \frac{1}{N}\sum_{t=1}^{N}(y(t) - \varphi(t)^T\theta)^2. \end{aligned} \tag{6}$$

The Least Squares Method (LSM) can find the following $\hat{\theta}$ that minimizes the estimation error $E_N(\theta, O_N)$:

$$\hat{\theta}_N = \left[\sum_{t=1}^{N}\varphi(t)\varphi(t)^T\right]^{-1}\sum_{t=1}^{N}\varphi(t)y(t). \tag{7}$$

Note that the ARX model can also be used to model the relationship between multiple inputs and multiple outputs [16], that is, we can have multiple flow intensities as the inputs and/or outputs in (1). For simplicity, here, we only model and analyze the correlation between two flow intensities.

There are several criteria to evaluate how well the learned model fits the real observation. Here, we use the following equation to calculate a normalized fitness score for model validation:

$$F(\theta) = \left[1 - \sqrt{\frac{\sum_{t=1}^{N}|y(t) - \hat{y}(t|\theta)|^2}{\sum_{t=1}^{N}|y(t) - \bar{y}|^2}}\right] \cdot 100, \tag{8}$$

where $\bar{y}$ is the mean of the real output $y(t)$. Basically, (8) introduces a metric to evaluate how well the learned model approximates the real data. A higher fitness score indicates that the model fits the observed data better, and its upper bound is 100. Given the observation of two flow intensities, we can always use (7) to learn a model even if this model does not reflect their relationship at all. Therefore, only a model with a high fitness score is really meaningful in characterizing the data relationship. We can set a range for the order $[n, m, k]$ rather than a fixed number to learn a list of model candidates, and then, the model with the highest fitness score can be chosen from them to characterize the data relationship. Note that we use the ARX model to learn the long-run relationship between two measurements, that is, a model $y = f(x)$ only captures the main characteristics of their relationship. The precise relationship between two measurements should be represented with $y = f(x) + \epsilon$, where ϵ is the modeling error. Note that ϵ is very small for a model with a high fitness score. This is one reason why we regard such models as likely invariants. In this paper, we choose a threshold of fitness scores to control the level of modeling errors ϵ.

5 Invariant Extracting Algorithm

Given the model template shown in (1), we analyzed how to automatically learn a model instance between two flow intensities. We may collect many flow intensity measurements from a complex system, but obviously, not any pairs of them would have such linear relationships. Meantime, due to system dynamics and uncertainties, some learned models may not be robust along time. Then, a challenging question is how to extract invariants from a large number of flow intensity measurements. In practice, we may build some relationships based on prior system knowledge. However, we believe that this knowledge is very limited and system dependent. In this section, we introduce our previous invariant extracting algorithm—FullMesh algorithm—and then discuss its computational complexity.

```
**FullMesh Algorithm**

Input: $I_i(t)$, $1 \le i \le n$
Output: $M_k$ and $p_k(\theta)$ for each time window $k$

Part I: Invariant Search
at time $t = l$ (i.e., $k = 1$),
    for each $I_i$ and $I_j$, $1 \le i, j \le n$, $i \ne j$
        learn a model $\theta_{ij}$ using Equation (7);
        compute $F_1(\theta_{ij})$ with Equation (8);
        if $F_1(\theta_{ij}) > \widetilde{F}$, then
            set $M_1 = M_1 \cup \{\theta_{ij}\}$ and $p_1(\theta_{ij}) = 1$.

Part II: Invariant Validation
for each time $t = k \cdot l$, $k > 1$,
    for each $\theta_{ij} \in M_k$,
        compute the $F_k(\theta_{ij})$ with Equation (8) using
            $I_i(t)$ and $I_j(t)$, $(k-1) \cdot l + 1 \le t \le k \cdot l$;
        update $p_k(\theta_{ij})$ with Equation (10);
        if $p_k(\theta_{ij}) \le P$ and $k \ge K$,
            then remove $\theta_{ij}$ from the $M_k$.
    output $M_k$ and $p_k(\theta)$.
    $k = k + 1$.
```

Fig. 2. Invariant extracting algorithm.

Assume that we have n measurements denoted by I_i, $1 \le i \le n$. Since we have little knowledge about their relationships in a specific system, we try any combination of two measurements to construct a model first and then continue to validate whether this model fits with the new monitoring data, that is, we construct all hypotheses of invariants first and then sequentially test the validity of these hypotheses in operation. Note that we always have sufficient monitoring data from an operational system to validate these hypotheses. The fitness score $F_i(\theta)$ given by (8) is used to evaluate how well a learned model matches the data observed during the ith time window. We denote the length of this window by l, that is, the window includes l sampling points. Further, we select a threshold $\widetilde{F}$, and the following piecewise function is used to determine whether a model fits the data:

$$f(F_i(\theta)) = \begin{cases} 1 & \text{if } F_i(\theta) > \widetilde{F} \\ 0 & \text{if } F_i(\theta) \le \widetilde{F}. \end{cases} \tag{9}$$

After receiving monitoring data for k of such windows, that is, the total $k \cdot l$ sampling points, we can calculate a confidence score with the following equation:

$$\begin{aligned} p_k(\theta) &= prob(F_t(\theta) > \widetilde{F}) \\ &= \frac{\sum_{i=1}^{k} f(F_i(\theta))}{k} \\ &= \frac{p_{k-1}(\theta) \cdot (k-1) + f(F_k(\theta))}{k}. \end{aligned} \tag{10}$$

Denote the set of valid models at time $t = k \cdot l$ by M_k, that is, $M_k = \{\theta | p_k(\theta) > P\}$. P is the confidence threshold we choose to determine whether a model has the potential to be an invariant. The invariant extracting algorithm is shown in Fig. 2, which consists of two parts: invariant search and

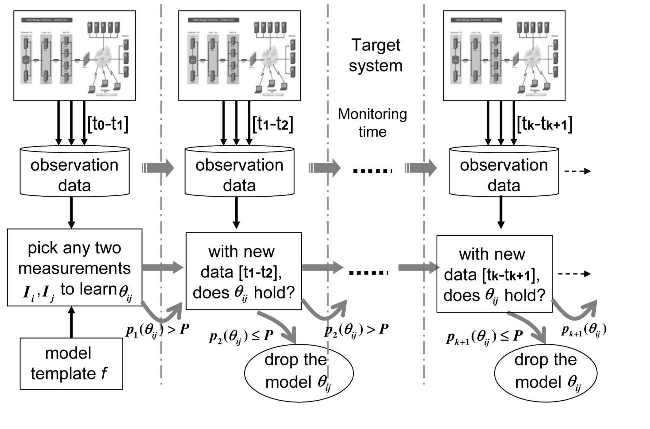

Target system
Monitoring time
[t0-t1]
[t1-t2]
[tk-tk+1]
observation data
observation data
observation data
pick any two measurements I_i, I_j to learn θ_{ij}
with new data [t1-t2], does θ_{ij} hold?
with new data [tk-tk+1], does θ_{ij} hold?
model template f
$p_1(\theta_{ij}) > P$
$p_2(\theta_{ij}) \le P$
$p_2(\theta_{ij}) > P$
$p_{k+1}(\theta_{ij}) \le P$
$p_{k+1}(\theta_{ij})$
drop the model θ_{ij}
drop the model θ_{ij}

Fig. 3. The process of invariant validation.

invariant validation. Part I starts to build a model for any two measurements and, then, Part II sequentially validates these models with the new monitoring data. After a time period ($K \cdot l$ sampling points and K is a selected number), if the confidence score of a model is less than the selected threshold P, we consider this model invalid and stop validating this model as an invariant. Theoretically, we should set $P = 1$ because a model can be regarded as an invariant only if this model holds all the time. However, since some rare incidents such as fault occurrences could affect normal relationships and deteriorate $p_k(\theta)$ temporally, we need several time windows of monitoring data to make sure that a model is invalid. The process of sequential validation is illustrated in Fig. 3.

It is straightforward to see that the computational complexity of the FullMesh algorithm mainly results from Part I of the algorithm. Given n flow intensity measurements, Part I needs to construct $n(n-1)/2$ models as the candidates of invariants. Note that given two flow intensity measurements, logically, we do not know which one should be chosen as the input or output (that is, x or y in (1)) in complex systems. Therefore, in our algorithms, we construct two models (with reverse input and output) first but only choose the model with the higher fitness score as the invariant candidate. For convenience, in this paper, we consider this as one time of invariant search though we construct two models between two measurements. If two learned models have very different fitness scores, we must have constructed an AutoRegressive (AR) model rather than an ARX model. Since we are only interested in the strong correlation between two measurements, we filter out those AR models by requesting a high fitness score in both models. In addition, compared to the sequential validation process—Part II, the computational complexity of Part I is much high because it involves matrix inverse operation in (7). However, given monitoring data, in fact, Part I can run offline to construct all possible candidates of invariants so that the computational complexity of the FullMesh algorithm should not be a serious issue for many problems. In practice, we can also use prior knowledge such as the system architecture to split all measurements into several subsystems and only search invariants within each subsystem to reduce the computational complexity.

Though Part I of the FullMesh algorithm can run offline to search invariants, it still may not scale well in large systems with thousands of measurements. We have been testing invariant-based solutions on several large information systems and networks. We notice that it can take days to search invariants from a huge amount of monitoring data though Part II of the algorithm runs very fast. To this end, we propose two efficient randomized algorithms to search invariants, which can be used to replace Part I of the FullMesh algorithm. However, these two algorithms are approximate algorithms with different levels of approximation in accuracy. Given n measurements, theoretically, we will need $n(n-1)/2$ searches to extract all possible invariants, and any reduction of search complexity may result in the loss of invariants. Therefore, our approximate algorithms may not be able to discover all invariants though their computational complexity can be significantly reduced. Note that we usually extract a large number of invariants from large systems, and in fact, these invariants include much redundancy in characterizing systems. Therefore, a little loss of invariants may not affect system management tasks at all, that is, in practice, we do not need the complete set of invariants most of the time. In addition, sometimes, we have to trade off accuracy with computational time because some systems evolve quickly (for example, adding and removing a server, installing new software, and so forth), and we want to extract invariants quickly and use them for system management tasks long before the system itself changes and the invariant set becomes invalid.

6 Use of Invariants

Large-scale distributed systems such as Internet services could consist of thousands of components. These system components produce a large amount of monitoring data to track their operational status. However, it is hard to correlate such monitoring data effectively across distributed systems and along observation time for system management. We calculate flow intensities from monitoring data collected at various points across distributed systems. Further, we derive invariant relationships between these measurements, and the validity of such invariants is not affected by the dynamics of user loads. Each invariant enables us to model some local properties of its related system components, and the combination of many invariants could well characterize the whole complex system. Therefore, we can interpret the system operational status in real time by tracking the changes of these invariants.

In previous work [14], [30], we proposed to track abrupt changes of invariants for fault detection in complex systems. After we extract invariants from distributed systems, model-based FDI methods are applied to track invariants in real time for anomaly detection. Model-based FDI methods have been widely analyzed in control theory [12], [9]. As discussed in Section 4, flow intensities x and y are measured, respectively, at the input and output of a system component. Assume that we derive an invariant relationship $y = f(x)$ to characterize the correlation between these two measurements. At time t, we use y_t to represent the actual observed output and use $\overline{y}_t$ to represent the simulated output from the invariant, that is, $\overline{y}_t = f(x_t)$, where x_t is the real observed input. Therefore, we should always have the

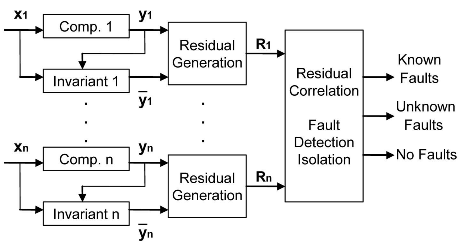


Fig. 4. Invariant-based FDI solution.

residual $R_t = |y_t - \overline{y}_t| \le \epsilon_M$, where ϵ_M is the threshold of modeling error. If a fault occurs inside this component, it may affect the flow relationship, and this invariant is likely to be violated. Therefore, we could detect such a fault in real time by tracking whether the real output stays in the same trajectory as the invariant model expects, that is, at time t, check whether $R_t \le \epsilon_M$. The invariant-based FDI solution is illustrated in Fig. 4.

Based on various physical meanings of flow intensity measurements, invariants could also characterize target systems from many different perspectives. With the combination of many invariants extracted from distributed systems, we could detect a wide class of faults in complex systems, including operator faults, software faults, hardware faults, and networking faults. Based on the dependency relationship between invariants and their monitoring components, we could also isolate faults by correlating broken invariants. In addition, the combination of invariants can also be used to capture, index, and retrieve the system history. For example, we can use a binary vector to represent the status of all invariants (broken or not) and further use this vector to build profiles or signatures for various faults. Later, these signatures can be considered as the symptoms to index and retrieve previous solutions from the system management history [5].

Note that in this section, we only use FDI as an example to illustrate the use of such invariants. Besides this use, invariants can also be used to support many other system management tasks such as service profiling, performance debugging, capacity planning, resource optimization, and situation awareness. Readers are encouraged to see more applications of such invariants and their experimental results in related papers [14], [13].

7 SmartMesh Algorithm

In this section, we introduce an approximate algorithm named SmartMesh to search invariants sequentially. Rather than searching all possible $n(n-1)/2$ relationships (a meshed graph), the SmartMesh algorithm sequentially extracts invariants from small clusters that are determined by search results from previous steps. The SmartMesh algorithm is illustrated in Fig. 5. Note that this approximate algorithm is only proposed to replace Part I of the FullMesh algorithm, and the invariant validation part remains the same as in the FullMesh algorithm. Here, we denote the set of all measurements as $C = \{I_i\}$, $1 \le i \le n$ and an empty set as ϕ, respectively.

At the first step, the SmartMesh algorithm randomly selects one measurement I_i from the whole set C and then starts to search all other measurements that may have relationships with this reference measurement I_i. After the first step (with $n-1$ searches), basically, we can split the whole set of measurements into two clusters: one cluster W_1 that includes all measurements that have relationships with the measurement I_i and another cluster $C - W_1$ that includes the remaining measurements that do not have relationships with I_i. Since all measurements in W_1 have linear relationships with the reference measurement I_i, they are very likely to have linear relationships between each other so that we search all possible invariants within this cluster. Conversely, since all measurements in $C - W_1$ do not have linear relationships with the reference measurement I_i, they are unlikely to have linear relationships with other measurements in W_1. In fact, this is the reason why we call I_i as the reference measurement. Based on the outcome of the first step, we will not search any relationships between measurements that belong to different clusters. For the following steps, we repeat the same process, that is, randomly select one measurement as a reference measurement and further split the remaining clusters C_s to two smaller clusters until eventually C_s becomes empty. Note that $\sum_s |W_s| = |C| = n$, where $|.|$ represents the size of a set. If one measurement does not have relationships with any other measurements in C, we consider this single measurement as a cluster with a size equal to one.

Let us give an example here to illustrate how this algorithm works. Assume that we have a set with five measurements $C = \{a, b, c, d, e\}$, and the set can be finally split into two smaller clusters with $\{a, b, c\}$ and $\{d, e\}$. The measurements within the same clusters have relationships and none otherwise. For the first step, assume that we

SmartMesh Algorithm

```
Input: $I_i(t)$, $1 \le i \le n$
Output: $M_1$ and $p_1(\theta)$

at time $t = l$ (i.e., $k = 1$), set $C_1 = C$, $M_1 = \phi$ and $s = 1$.
do
  randomly select one measurement $I_i$ from $C_s$;
  set $W_s = \{I_i\}$;
  for each $I_j \in C_s$, $j \neq i$
    learn a model $\theta_{ij}$ using Equation (7);
    compute $F_1(\theta_{ij})$ with Equation (8);
    if $F_1(\theta_{ij}) > \widetilde{F}$, then
      set $M_1 = M_1 \cup \{\theta_{ij}\}$, $p_1(\theta_{ij}) = 1$, $W_s = W_s \cup \{I_j\}$.

  for each $I_u, I_v \in W_s - \{I_i\}$, $u \neq v$
    learn a model $\theta_{uv}$ using Equation (7);
    compute $F_1(\theta_{uv})$ with Equation (8);
    if $F_1(\theta_{uv}) > \widetilde{F}$, then
      set $M_1 = M_1 \cup \{\theta_{uv}\}$, $p_1(\theta_{uv}) = 1$.

  $C_{s+1} = C_s - W_s$.
  $s = s + 1$.
while $C_s$ is not empty.
return $M_1$ and $p_1(\theta)$.
```

Fig. 5. SmartMesh algorithm.

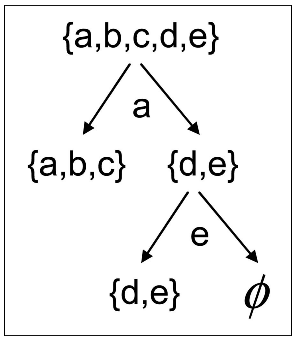


Fig. 6. An example of the SmartMesh algorithm.

randomly pick one measurement, a, from the set C as the reference measurement. After modeling relationships with the other four measurements (four searches), we know that a only has relationships with b and c. Since both b and c have linear relationships with a, they are likely to have a linear relationship too so that we also model the relationship between b and c. Conversely, $\{d, e\}$ do not have linear relationships with a so that they are unlikely to have linear relationships with $\{b, c\}$. Therefore, we do not search any relationships between $\{d, e\}$ and $\{b, c\}$. Then, for the second step, we have to randomly pick another measurement from the remaining set $C_2 = \{d, e\}$ as the reference measurement and assume that this time we select e. Then, with another search, we know that e has a relationship with d, and C_3 becomes empty after the first two steps. This invariant search process is illustrated in Fig. 6.

Now, let us consider a triangle relationship among three measurements $\{a, b, c\}$ so as to understand the basic concept underlying the SmartMesh algorithm. In one case, assume that a has linear relationships with b and c, that is, $a = f(b)$ and $c = g(a)$, where $f(.)$ and $g(.)$ are linear functions as shown in (1). Therefore, we can conclude the linear relationship between b and c as $c = g(f(b))$. As mentioned earlier, in each invariant search, we always construct two models with reverse input and output between two measurements. If both functions $f(.)$ and $g(.)$ are linear, it is easy to prove that the function $g(f(.))$ is also linear, which can be characterized with the ARX model shown in (1). However, as mentioned in Section 4, since there exist modeling errors in both $a = f(b)$ and $c = g(a)$, the derived equation $c = g(f(b))$ may not be precise enough to characterize the relationship between b and c. Therefore, we model their relationship directly with the observed data. In the second case, assume that a has a relationship with b but not c; then, it is very unlikely that b and c will have a strong relationship. Otherwise, as analyzed in the first case, theoretically, we could conclude a linear relationship between a and c, which contradicts our assumption. However, in engineering practice, there always exist modeling errors in these equations. We select a threshold $\widetilde{F}$ and use (9) to determine whether two measurements have a relationship or not. For the second case, if the correlation between a and c has a fitness score slightly below the threshold, the correlation between b and c could still have a fitness score slightly higher than the threshold. Therefore, the second case could still happen due to modeling errors ϵ in (1). Note that it is impossible to get the second case if $\epsilon = 0$. Since the SmartMesh algorithm does not search relationships between measurements that belong to different clusters, it may not discover as many invariants as the FullMesh algorithm does. This is the reason why we consider the SmartMesh algorithm as an approximate algorithm. The accuracy of the SmartMesh algorithm will be analyzed in Section 9. With loss of accuracy, the SmartMesh algorithm could significantly reduce the computational complexity of invariant search. Denote the computational complexity of Part I in the FullMesh algorithm as T_1, which includes $n(n-1)/2$ searches, that is, $T_1 = n(n-1)/2$. Denote the computational complexity of the SmartMesh algorithm as T_2. Given n measurements, assume that the set of measurements C can be split into K clusters $W_s (1 \le s \le K)$, as shown in the SmartMesh algorithm. For convenience, denote the sizes of clusters W_s as m_s, respectively, that is, $m_s = |W_s|$, and we should have $\sum_{s=1}^{K} m_s = |C| = n$. For the first step, the randomly selected reference measurement needs $n - 1$ searches to discover its cluster W_1 and, then, we need $(m_1 - 1)(m_1 - 2)/2$ searches to discover invariants within this cluster W_1. Therefore, we can calculate the computational complexity of the first step with the following equation:

$$\begin{aligned} T_2(1) &= n - 1 + \frac{(m_1 - 1)(m_1 - 2)}{2} \\ &= n - m_1 + \frac{m_1(m_1 - 1)}{2}. \end{aligned} \tag{11}$$

Now, for the second step, the second reference measurement needs $n - m_1 - 1$ searches to discover its cluster W_2 and, then, we need $(m_2 - 1)(m_2 - 2)/2$ searches to discover invariants within this cluster W_2. Therefore, we can obtain the computational complexity of the second step, $T_2(2)$, by replacing m_1 with m_2 and replacing n with $n - m_1$ in (11), respectively. For all K steps, we can conclude the computational complexity of the SmartMesh algorithm with the following equation:

$$\begin{aligned} T_2 &= \sum_{s=1}^{K} T_2(s) \\ &= \sum_{s=1}^{K} \frac{m_s(m_s - 1)}{2} + \sum_{s=1}^{K} \left(n - \sum_{j=1}^{s} m_j \right) \\ &= \sum_{s=1}^{K} \frac{m_s(m_s - 1)}{2} + \sum_{s=1}^{K} (s - 1) m_s, \end{aligned} \tag{12}$$

where $n = \sum_{j=1}^{K} m_j$.

SmartMesh is a randomized algorithm [6], [18] because the sequential order of clusters W_s is randomly determined by the selection of reference measurements and the sequential order of cluster sizes could affect the complexity T_2 significantly. Given a sequential order of cluster sizes $(m_1, m_2, \cdots, m_K)$, we can use (12) to calculate its computational complexity $T_2(m_1, m_2, \cdots, m_K)$. For K clusters, in total, we could have $K!$ (factorial) of such sequential orders. For convenience, we use $o_i (1 \leq i \leq K!)$ to represent a sequential order of cluster sizes, that is, $o_i = Permutation(\{m_s\}_{1 \leq s \leq K})$. Note that the sequential order of cluster sizes o_i only affects the second item at the right side of (12) but not the first item. Given n measurements that can be split into K clusters, if the cluster sizes monotonely decrease at each step, that is, $m_1 \geq m_2 \geq \cdots \geq m_K$, we have the smallest T_2 (the best case). Conversely, if the cluster sizes monotonely increase at each step, we have the largest T_2 (the worst case).

In fact, since we randomly (uniform) select reference measurements at each step and bigger clusters include more measurements, the probability to choose a reference measurement from big clusters is higher than that from small clusters. Therefore, T_2 in the average case should be biased toward the smallest T_2 of the best case. Given a sequential order of cluster sizes $(m_1, m_2, \cdots, m_K)$, we can calculate its probability with the following equation according to Bayesian rules [7]:

$$\begin{aligned} & P(m_1, \cdots, m_K) \\ = \; & P(m_1)P(m_2|m_1)\cdots P(m_K|m_1, \cdots, m_{K-1}) \\ = \; & \frac{m_1}{n}\frac{m_2}{n-m_1}\cdots\frac{m_K}{m_K} \\ = \; & \prod_{j=1}^{K}\frac{m_j}{n-\sum_{i=1}^{j-1} m_i}. \end{aligned} \tag{13}$$

Now, we can calculate the computational complexity of the average case with the following equation:

$$E(T_2) = \sum_{i=1}^{K!} T_2(o_i)P(o_i), \tag{14}$$

where $T_2(o_i)$ and $P(o_i)$ are given in (12) and (13), respectively. Currently, we do not have a compact form of this average computational complexity $E(T_2)$. One problem is that $E(T_2)$ is not only dependent on the number of input measurements n but also the number of clusters K and the distribution of cluster size $\{m_s\}_{1 \leq s \leq K}$. In other words, $E(T_2)$ is dependent on not only the size of input data but also the "content" of input data. Note that here, we do not make assumptions on these parameters or distributions for our average case analysis because such assumptions could result in a large estimation error for a specific problem or a specific set of measurements. In our future work, we will investigate whether we can derive a tight bound for $E(T_2)$. In our experiments shown in Section 9, we run our algorithms many times to analyze the average case for a given set of measurements. According to (11), the computational complexity T_2 becomes smaller if we search invariants from bigger clusters first. In practice, we may have prior knowledge to determine which measurements are more likely to be from big clusters. For example, as shown in Section 9, those measurements responding to workloads directly are likely to have many relationships between each other and form a big cluster. In the implementation of our algorithms, we can increase the probability of these nodes as reference points rather than the uniform selection. Meantime, if a reference point does not find any relationships after a threshold of trials (for example, 5 percent of all needed searches for this node), we can stop the current search process and choose another node as a new reference point. However, this may increase the complexity in our algorithm implementation because those tested relationships have to be maintained after we choose a new reference point. According to (14), in the above cases, we can reduce the computational complexity of the average case by increasing the $P(o_i)$ for those o_i with small $T_2(o_i)$.

However, we always have $T_2 \leq T_1$. Given a sequential order of cluster sizes $(m_1, m_2, \cdots, m_K)$, the reduction of computational complexity can be calculated with the following equation:

$$\begin{aligned} T_1 - T_2 &= \frac{n(n-1)}{2} - \sum_{s=1}^{K}\frac{m_s(m_s-1)}{2} - \sum_{s=1}^{K}\left(n - \sum_{j=1}^{s} m_j\right) \\ &= \frac{n(n-1)}{2} - \sum_{s=1}^{K}\frac{m_s^2}{2} + \sum_{s=1}^{K}\frac{m_s}{2} - \sum_{s=1}^{K}\left(n - \sum_{j=1}^{s} m_j\right) \\ &= \frac{n^2}{2} - \sum_{s=1}^{K}\frac{m_s^2}{2} - \sum_{s=1}^{K}\left(n - \sum_{j=1}^{s} m_j\right) \\ &= \frac{\left(\sum_{s=1}^{K} m_s\right)^2 - \sum_{s=1}^{K} m_s^2}{2} - \sum_{s=1}^{K}\left(n - \sum_{j=1}^{s} m_j\right) \\ &= \sum_{s=1}^{K}\left[m_s\left(n - \sum_{j=1}^{s} m_j\right)\right] - \sum_{s=1}^{K}\left(n - \sum_{j=1}^{s} m_j\right) \\ &= \sum_{s=1}^{K}(m_s - 1)\left(n - \sum_{j=1}^{s} m_j\right). \end{aligned} \tag{15}$$

Since $m_s \geq 1$ and $s \leq K$, we always have

$$n = \sum_{j=1}^{K} m_j \geq \sum_{j=1}^{s} m_j.$$

Therefore, we should always have $T_1 - T_2 \geq 0$. The value of $T_1 - T_2$ is also dependent on the sequential order of cluster sizes. Let us consider two special cases here: In the first case, if the set of n measurements is split into $K = n$ clusters with the size of each cluster $m_s = 1$ (that is, there is no relationship between any measurements), according to (15), we have $T_1 = T_2$; in the second case, if the set of n measurements is split into $K = 1$ cluster with the size of this cluster $m_1 = n$ (that is, there is a relationship between any pair of measurements), we also have $T_1 = T_2$. Both cases are the worst-case scenarios, which are very rare in practice. For all other cases, according to (15), T_2 should be much smaller than T_1.

Since T_2 depends on many unknown parameters including the number of measurements n, the number of clusters K, and the distribution of cluster size $\{m_s\}_{1 \leq s \leq K}$, in general, it is very difficult to derive the best-case scenarios. Given a set of clusters, as discussed earlier, we have the smallest T_2 if the cluster sizes monotonely decrease at each step. However, even if parameters n and K are fixed, the problem of minimizing T_2 is still as hard as an integer programming problem [27], which is often NP-hard. Thus, let us consider a special case when the sizes of all clusters are equal, that is, $m_s = \frac{n}{K}$, $1 \leq s \leq K$. With this assumption, now, T_2 only depends on two parameters n and K, and (12) can be rewritten as

$$\begin{aligned} T_2 &= \sum_{s=1}^{K} \frac{m_s(m_s-1)}{2} + \sum_{s=1}^{K}(s-1)m_s \\ &= \frac{n(K^2-2K+n)}{2K}. \end{aligned} \tag{16}$$

For convenience in our formula deduction, let us consider K as a continuous variable. Then, we can derive the derivative of T_2 with respect to K with the following equation:

$$\frac{dT_2}{dK} = \frac{nd(K^2-2K+n)}{2dK} = \frac{n(K^2-n)}{2K^2}. \tag{17}$$

According to (17), we have $\frac{dT_2}{dK}=0$ only if $K=\sqrt{n}$. Meantime, $\frac{dT_2}{dK}<0$ if $K<\sqrt{n}$, and $\frac{dT_2}{dK}>0$ if $K>\sqrt{n}$. As discussed earlier, when $K=1$ or $K=n$, we have maximal $T_2=T_1$. Therefore, we have minimal T_2 when $K=\sqrt{n}$ (in practice, K should be the integer closest to $\sqrt{n}$) and can reduce the computational complexity of T_2 from $O(n^2)$ to $O(n\sqrt{n})$ in this case. Without the assumption on equal size clusters, note that the T_2 of the best case should have lower computational complexity than $O(n\sqrt{n})$.

8 SimpleTree Algorithm

In this section, we introduce another approximate algorithm named SimpleTree for invariant search. Compared to the SmartMesh algorithm, the SimpleTree algorithm can further reduce computational complexity but may also compromise approximation accuracy in invariant search. Note that the SimpleTree algorithm is also proposed to replace Part I of the FullMesh algorithm. As discussed in Section 7, let us consider a triangle relationship among three measurements $\{x,y,z\}$. As a time series, assume that x has already built models with y and z, respectively. For accuracy, the SmartMesh algorithm uses (7) and monitoring data to learn a model between y and z. Conversely, based on the other two models available in the triangle relationship, the SimpleTree algorithm directly derives a mathematical model between y and z without using any monitoring data.

For convenience, we introduce the backward shift operator q^{-1} by $q^{-1}y(t)=y(t-1)$. Therefore, we can rewrite the ARX model in (1) as

$$y(t) = x(t)\frac{b_0+b_1q^{-1}+\cdots+b_mq^{-m}}{1+a_1q^{-1}+\cdots+a_nq^{-n}}. \tag{18}$$

Without loss of generality, here, we assume that $k=0$ in (1). If $k>0$, we set the first k coefficients $b_i=0 (0\le i\le k-1)$. Let us denote $a_0=1$, $A_n=(a_0,a_1,\cdots,a_n)$, $B_m=(b_0,b_1,\cdots,b_m)$, and $Q_k=(q^0,q^{-1},\cdots,q^{-k})$. Note that A_n and B_m are coefficient vectors. According to the definition of θ in (2), we should have $(1,\theta^T)=(A_n,B_m)$. Therefore, we can rewrite (18) with

$$y(t) = x(t)\frac{B_mQ_m^T}{A_nQ_n^T}, \tag{19}$$

where Q^T is the matrix transposition of Q. In a similar way, assume that the measurements x and z have the following relationship:

$$x(t) = z(t)\frac{D_{\widehat{m}}Q_{\widehat{m}}^T}{C_{\widehat{n}}Q_{\widehat{n}}^T}, \tag{20}$$

where $D_{\widehat{m}}$ and $C_{\widehat{n}}$ are coefficient vectors. As mentioned earlier, in each invariant search, we always construct two models with reverse input and output between two measurements. Thus, based on (19) and (20), we can derive the following model between y and z:

$$y(t) = z(t)\frac{Q_mB_m^TD_{\widehat{m}}Q_{\widehat{m}}^T}{Q_nA_n^TC_{\widehat{n}}Q_{\widehat{n}}^T} \tag{21}$$

where $B_m^TD_{\widehat{m}}$ is an $(m+1)\times(\widehat{m}+1)$ matrix, and $A_n^TC_{\widehat{n}}$ is an $(n+1)\times(\widehat{n}+1)$ matrix. For convenience, denote $V=B_m^TD_{\widehat{m}}$ and $U=A_n^TC_{\widehat{n}}$.

According to the rules of polynomial multiplication [24], we can rewrite (21) with the following equation:

$$y(t) = z(t)\frac{F_{m+\widehat{m}}Q_{m+\widehat{m}}^T}{E_{n+\widehat{n}}Q_{n+\widehat{n}}^T}, \tag{22}$$

where $F_{m+\widehat{m}}$ and $E_{n+\widehat{n}}$ are $m+\widehat{m}+1$-dimensional and $n+\widehat{n}+1$-dimensional coefficient vectors, respectively. Denote the jth$(0\le j\le m+\widehat{m})$ element of $F_{m+\widehat{m}}$ as $F_{m+\widehat{m}}^j$ and the lth$(0\le l\le n+\widehat{n})$ element of $E_{n+\widehat{n}}$ as $E_{n+\widehat{n}}^l$. Based on (21) and (22), we can derive the following equations:

$$V = B_m^TD_{\widehat{m}}, \tag{23}$$

$$U = A_n^TC_{\widehat{n}}, \tag{24}$$

$$F_{m+\widehat{m}}^j = \sum_{i=0}^{j} V_{i,j-i}, \tag{25}$$

$$E_{n+\widehat{n}}^l = \sum_{k=0}^{l} U_{k,l-k}. \tag{26}$$

Note that for any $i>m$ or $j>\widehat{m}$, $V_{i,j}=0$, and for any $i>n$ or $j>\widehat{n}$, $U_{i,j}=0$.

In the triangle relationship among measurements $\{x,y,z\}$, given two models with parameters $(1,\theta_{yx}^T)=(A_n,B_m)$ and $(1,\theta_{xz}^T)=(C_{\widehat{n}},D_{\widehat{m}})$, we can calculate the matrices U and V first and then derive a model θ_{yz} with $(1,\theta_{yz}^T)=(E_{n+\widehat{n}},F_{m+\widehat{m}})$, where $F_{m+\widehat{m}}$ and $E_{n+\widehat{n}}$ are computed with (25) and (26). Based on the above discussion, we introduce the SimpleTree algorithm in Fig. 7. The SimpleTree algorithm is same as the SmartMesh algorithm except for the underlined part shown in Fig. 7. After randomly selecting a reference measurement and discovering its cluster W_s, the SimpleTree algorithm does not use (7) to learn models for measurements in the same cluster W_s. Instead, it mathematically derives these models from the learned models that correlates the reference measurement

```
SimpleTree Algorithm

Input: $I_i(t)$, $1 \leq i \leq n$
Output: $M_1$ and $p_1(\theta)$

at time $t = l$ (i.e., $k = 1$), set $C_1 = C$, $M_1 = \phi$ and $s = 1$.
do
    randomly select one measurement $I_i$ from $C_s$;
    set $W_s = \{I_i\}$;
    for each $I_j \in C_s$, $j \neq i$
        learn a model $\theta_{ij}$ using Equation (7);
        compute $F_1(\theta_{ij})$ with Equation (8);
        if $F_1(\theta_{ij}) > \widetilde{F}$, then
            set $M_1 = M_1 \cup \{\theta_{ij}\}$, $p_1(\theta_{ij}) = 1$, $W_s = W_s \cup \{I_j\}$.

    for each $I_u, I_v \in W_s - \{I_i\}$, $u \neq v$
        derive a model $\theta_{uv}$ using $\theta_{ui}$, $\theta_{iv}$,
            and Equations (23)-(26);
        compute $F_1(\theta_{uv})$ with Equation (8);
        if $F_1(\theta_{uv}) > \widetilde{F}$, then
            set $M_1 = M_1 \cup \{\theta_{uv}\}$, $p_1(\theta_{uv}) = 1$.

    $C_{s+1} = C_s - W_s$.
    $s = s + 1$.
while $C_s$ is not empty.
return $M_1$ and $p_1(\theta)$.
```

Fig. 7. SimpleTree algorithm.

with the other measurements in the cluster W_s. Note that for any two measurements in the same cluster, they form a triangle relationship with the reference measurement. Therefore, the SimpleTree algorithm only needs to search a two-level tree structure for each cluster, whose root is the reference measurement. This is the reason why we call this algorithm as the SimpleTree algorithm.

As discussed in Section 7, a derived model may not be as accurate as the model directly learned from monitoring data. The modeling errors of the two models (used to infer the third model) could convolve and then lead to a large error in the third model. Meantime, the order of a derived model is much larger than that of the learned model, that is, the derived model is not as compact as the learned model and may also overfit the monitoring data. For example, although θ_{yx} is an $n+m$-dimensional vector and θ_{xz} is an $\widehat{n}+\widehat{m}$-dimensional vector, the derived model θ_{yx} is an $n+m+\widehat{n}+\widehat{m}$-dimensional vector. However, in distributed information systems, the latency of user requests is very short, so the orders of ARX models (such as $[n, m]$ and $[\widehat{n}, \widehat{m}]$) are usually very small. For example, we set $0 \leq n$ and $m \leq 2$ to extract invariants in our experiments. Therefore, the order of a derived model (such as $[n+\widehat{n}, m+\widehat{m}]$) will remain small though it is larger than that of the other two models. Compared to the SmartMesh algorithm, the SimpleTree algorithm may lose more invariants because some derived models may not be accurate enough to pass the sequential testing process—Part II of the FullMesh algorithm. Section 9 includes experimental results about the accuracy of the SimpleTree algorithm.

Denote the computational complexity of the SimpleTree algorithm as T_3. As discussed earlier, since the model orders such as $[n, m]$ and $[\widehat{n}, \widehat{m}]$ are very small, the computational complexity of (23)-(26) is so small that it can be ignored in the analysis of T_3. In fact, (23)-(26) only involve several matrix multiplications. Conversely, if we use (7) to learn a model, we need at least N times of such matrix multiplications, as well as an operation of matrix inversion. Note that N is a big number because we usually need hundreds of data points to learn a model. At the first step of the SimpleTree algorithm, the randomly selected reference measurement needs $n-1$ searches to discover its cluster W_1 and, then, we derive analytical models between measurements within this cluster W_1. Therefore, we have the computational complexity of the first step: $T_3(1) = n-1$. For the second step, the second reference measurement needs $n - m_1 - 1$ searches to discover its cluster W_2. By following a similar analysis, for all K steps, we have the following equation to calculate T_3:

$$\begin{aligned} T_3 &= \sum_{s=1}^{K} T_3(s) \\ &= n - 1 + n - m_1 - 1 + \cdots + n - \sum_{s=1}^{K-1} m_s - 1 \\ &= K(n-1) - \sum_{s=1}^{K} (K-s) m_s \\ &= \sum_{s=1}^{K} s m_s - K. \end{aligned} \tag{27}$$

T_3 is also dependent on the sequential order of cluster sizes $\{m_s\}_{1 \leq s \leq K}$. Given n measurements that can be split into K clusters, if the cluster sizes monotonely decrease at each step, that is, $m_1 \geq m_2 \geq \cdots \geq m_K$, we have the smallest T_3 (the best case). Conversely, if the cluster sizes monotonely increase at each step, we have the largest T_3 (the worst case). As in Section 7, here, we use $o_i (1 \leq i \leq K!)$ to represent a sequential order of cluster sizes. For the average case, we have

$$E(T_3) = \sum_{i=1}^{K!} T_3(o_i) P(o_i), \tag{28}$$

where $T_3(o_i)$ and $P(o_i)$ are given in (27) and (13), respectively.

In practice, for a specific problem, we do not know how K (the number of clusters) increases with the growth of n (the number of measurements). If K is constant, the computational complexity of the SimpleTree algorithm T_3 would be $O(n)$; if K increases with $O(\lg(n))$, T_3 would be $O(n \lg(n))$. Therefore, T_3 is dependent on not only the size of input data n but also the number of clusters K included in this data. However, compared to T_2, it is straightforward to derive the reduction of computational complexity in T_3 with the following equation:

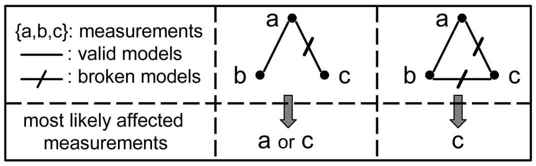


Fig. 8. Graph-based reasoning in fault isolation.

$$T_2 - T_3 = \sum_{s=1}^{K} (T_2(s) - T_3(s)) = \sum_{s=1}^{K} \frac{(m_s - 1)(m_s - 2)}{2}. \quad (29)$$

Since $m_s \geq 1 (1 \leq s \leq K)$, we always have $T_2 \geq T_3$. For the two special cases discussed in Section 7, if a set of n measurements is split into $K = n$ clusters with the size of each cluster $m_s = 1$, according to (29), we have $T_2 = T_3$ (the worst case); however, in the second case, if the set of n measurements is split into $K = 1$ cluster with the size of this cluster $m_1 = n$, we have $T_2 = n(n-1)/2$ and $T_3 = n - 1$ (the best case). In this case, we reduce the computational complexity of invariant search from $O(n^2)$ to $O(n)$. In general, according to (29), the SimpleTree algorithm should significantly reduce the computational complexity in invariant search though it could compromise approximation accuracy.

As discussed above, any two measurements in the same cluster W_s form a triangle relationship with their reference measurement and, in this relationship, we use two learned models to derive the third one. For some system management tasks, the third model may only provide "redundant" information because it can be derived from the other two models directly. In this case, we may only need to search the tree structure of invariants without any model derivation in the SimpleTree algorithm. However, for some system management tasks such as fault isolation, these derived models could support us to isolate faults more precisely. For example, in a triangle relationship, if we observe one of the two learned models is violated due to fault occurrences, we cannot determine which measurement (among the two measurements associated with the broken model) is affected. Instead, if we have three models and two models are violated, we could quickly determine which measurement is most likely affected. An example is given in Fig. 8 to illustrate such graph-based reasoning in fault isolation.

9 Experiments

Our invariant search experiments are performed in a typical three-tier Web system, which includes an Apache Web server, a JBoss application server, and a MySQL database server. Fig. 9 illustrates the architecture of our experimental system and its components. The application software running on this system is Pet store [23]. Pet store is a sample application written by Sun Microsystems to demonstrate how to use the Java 2 Enterprise Edition (J2EE) platform in developing flexible, scalable cross-platform e-commerce applications.

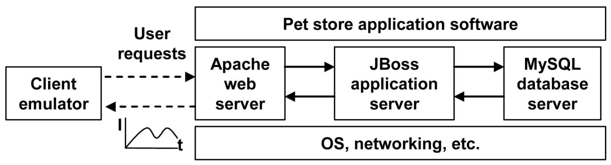


Fig. 9. The experimental system.

Category	Measurements	Web	AP	DB
CPU	utilization, user usage time, system usage time, idle time, IO wait time, IRQ time, soft IRQ time	7	7	7
Disk	# of write operations, # of write sectors, # of write merges, write time, IO time, # of pending IO operations, weighted IO time	7	7	7
OS	# of used file descriptors, # of max file descriptors, free physical memory size	2	3	2
Network	# of RX packets, # of RX bytes, # of RX errors, # of RX multicasts, # of TX packets, # of TX bytes	11	6	6
JVM	used heap memory size, processing time, # of live threads, peak # of threads, total # of threads	-	5	-
JBoss	# of processing EJBs, # of cached EJBs, # of pooled EJBs, # of created EJBs, # of DB connections, # of DB connections in use	-	6	-
Apache	# of http requests, # of http requests based on specific URL types	13	-	-
MySQL	# of SQL queries, # of SQL queries based on specific table types	-	-	15

Fig. 10. Categories of measurements.

Just like other Web services, here, users can visit the Pet store Web site to buy a variety of pets. We develop a client emulator to generate a large class of different user scenarios and workload patterns. Various user actions such as browsing items, searching items, account login, adding an item to a shopping cart, payment, and checkout are included in our workloads. A certain randomness of user behaviors is also considered in the emulated workloads. For example, a user action is randomly selected from all possible user scenarios that could follow the previous user action. The time interval between two user actions is also randomly selected from a reasonable time range. Note that workloads are dynamically generated with much randomness and variance so that we never get a similar workload twice in our experiments.

Monitoring data are collected from the three servers used in our testbed system. Many commercial and open source tools are available to instrument systems and collect a wide class of monitoring data. Fig. 10 shows the categories of our monitoring data. In total, we have eight categories, and each category includes different numbers of measurements. The number of measurements collected from each server is listed in the right three columns, which represent three servers, respectively. This monitoring data is used to calculate various flow intensities with a sampling unit equal to 6 sec. We have a total of 111 flow intensity measurements from three servers, and all of these measurements are used in the following experiments, that is, $n = 111$. In our experiments, we collect 1.5 hours of data to construct models and then continue to test these models for every half an hour, that is, the window size in the FullMesh algorithm is half an hour and includes 300 data points. Studies have shown that users are often dissatisfied if the latency of their Web requests is longer than 10 sec [11]. Therefore, the order of the ARX model (that is, $[n, m]$ shown in (1)) should have a very

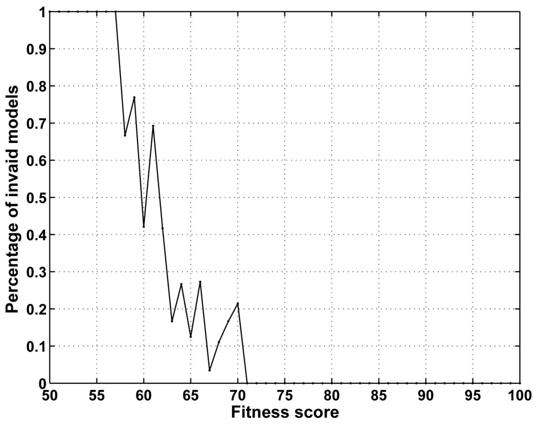


Fig. 11. Percentage of discarded models and their fitness scores.

narrow range. In our experiments, since the sampling time is selected to be 6 sec, we set $0 \leq n$ and $m \leq 2$.

By combining every two measurements from the total 111 measurements, Part I of the FullMesh algorithm in total builds 6,105 models (that is, $n(n-1)/2 = 6,105$) as invariant candidates. For each model, a fitness score is calculated according to (8). In our experiments, we select the threshold of fitness score $\widetilde{F} = 50$. In practice, we observe that those models with fitness scores over 50 usually illustrate strong linear correlations between two measurements. All our algorithms are implemented in Java programming language and run on a Pentium 4 machine with a 3.0-GHz CPU and 512-Kbyte L2 cache. It takes 1,296 sec of execution time for Part I of the FullMesh algorithm to process 1.5 hours of monitoring data and construct all 6,105 models. We observe that 1,158 models (among the total 6,105 models) have fitness scores higher than 50. As mentioned earlier, Part II of the FullMesh algorithm runs very fast, and it only takes 2.5 sec of execution time on the average for each sequential testing phase to process half an hour of data and validate 1,158 models. In our experiments, we set the threshold of confidence scores $P = 1$, that is, a model will be immediately discarded if it fails the validity test in one time window. Finally, after seven phases of sequential testings with various workloads, eventually, we have 975 valid models left, and this number becomes quite stable in the following sequential testing phases. Fig. 11 shows the percentage of models with different fitness scores that are discarded in the following sequential testing process—Part II. From the figure, we observe a sort of "phase transition" phenomenon. All models with fitness scores lower than 57 in Part I are 100 percent discarded in Part II. Conversely, all models with fitness scores over 70 in Part I continue to remain in Part II. For those models with scores between 57 and 70 in Part I, some of them are discarded in Part II, whereas others pass the sequential testing process. Therefore, we believe that the score 50 is a good threshold for our invariant extraction algorithms. Here, we consider the remaining 975 models as the invariants of the testbed system. For example, if we use I_{ejb}, I_{jvm}, and I_{sql} to represent the flow intensities of the "number of Enterprise JavaBeans (EJB) created," the "Java virtual machine (JVM) processing time," and the "number of MySQL queries" measured from the testbed system, we extract the following

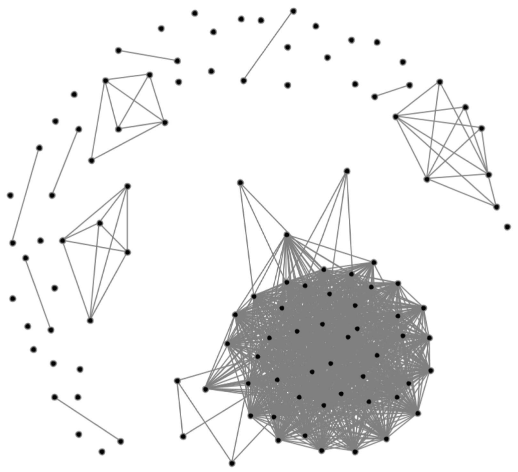


Fig. 12. Network of measurements and invariants.

invariants among these measurements. More details about these experiments can be seen in our previous work [13]:

$$I_{ejb}(t) = 0.07 I_{ejb}(t-1) - 0.57 I_{jvm}(t), \tag{30}$$

$$I_{sql}(t) = 0.34 I_{sql}(t-1) + 1.41 I_{ejb}(t) + 0.2 I_{ejb}(t-1). \tag{31}$$

With 111 flow intensity measurements, we eventually discover 975 likely invariants from them. Fig. 12 illustrates how these invariants are distributed across these measurements. This figure is plotted with an open source software library named JUNG [19]. In this figure, a node represents a measurement, whereas an edge represents an invariant relationship between the connected two nodes. Therefore, we have 111 nodes and 975 edges in this figure. From this figure, we notice that the 111 measurements can be split into many clusters. The biggest cluster in this figure includes 52 measurements. These measurements respond to user loads directly so that they have many relationships between each other. Meantime, we also observe several smaller clusters and many isolated nodes. These isolated nodes (29 nodes) represent those measurements that do not have relationships with any other measurements.

The SmartMesh algorithm is applied on the same monitoring data to search invariants. Since the SmartMesh algorithm is a randomized algorithm and its computational complexity is dependent on the sequential order of the cluster sizes, as shown in Fig. 12, we run the algorithm 15 rounds to demonstrate its accuracy and efficiency. Table 1 illustrates the results of the SmartMesh algorithm. The first column "No." refers to the 15 rounds of executions. The second column "initial invariants" refers to the number of invariants resulting from the SmartMesh algorithm, and the third column "final invariants" refers to the number of invariants left after the same seven sequential testing phases used in the above experiments of the FullMesh algorithm. Note that all "final invariants" are extracted from the set of "initial invariants" after they pass the sequential testing phases. The last column "time" represents the execution time (in seconds) used in each round of executions. The last row in Table 1 shows the average number of extracted invariants and the average execution time for each round.

TABLE 1
Results of the SmartMesh Algorithm

No.	initial invariants	final invariants	time(s)
1	984	889	422
2	1135	968	458
3	1140	972	454
4	988	932	429
5	1128	972	440
6	1140	972	458
7	1137	968	493
8	1127	972	446
9	964	868	446
10	1070	969	450
11	1141	973	428
12	1135	968	457
13	1071	969	426
14	1140	972	452
15	1055	891	432
avg.	**1090**	**950**	**446**

As shown in Table 1, each round of execution may result in a different number of initial invariants and different execution time. As discussed in Section 7, at each step, we randomly choose reference measurements to search clusters so that the results of the SmartMesh algorithm are affected by this random process. Compared to the 1,158 initial invariants extracted in the FullMesh algorithm, the SmartMesh algorithm always results in fewer invariants. As analyzed in Section 7, this is because the SmartMesh algorithm does not search invariants between measurements that belong to different clusters. For example, in a triangle relationship among measurements $\{a, b, c\}$, assume that a has a relationship with b but not c; the SmartMesh algorithm will not search the relationship between b and c because they are unlikely to have a strong relationship. Due to modeling errors, we may lose such relationships so that we often observe fewer initial invariants in the SmartMesh algorithm. More details are discussed in Section 7. However, such lost relationships are usually weak so that they are unlikely to pass the following sequential testing phases. Otherwise, if b and c have a strong relationship, we can conclude that a should have a strong relationship with c too, which contradicts our earlier assumption. Therefore, though we extract fewer initial invariants in the SmartMesh algorithm, the number of final invariants left after the sequential testing phases may still be very close to the 975 final invariants resulting from the FullMesh algorithm. The number of initial invariants is not so important as the number of the final invariants because only the final invariants are really used to characterize complex systems and support system management tasks. Note that the set of invariants extracted by the SmartMesh algorithm is always a subset of invariants extracted by the FullMesh algorithm because both algorithms use the same equation (that is, (7)) to learn models, but SmartMesh algorithm does not search relationships between all possible pairs.

Fig. 13 illustrates the performance comparison between the SmartMesh algorithm and the FullMesh algorithm. For each round, the "number of initial invariants," the "number of final invariants" and the "execution time" in Table 1 are divided by the same metrics of the FullMesh algorithm, respectively, that is, 1,158 initial invariants, 975 final invariants, and 1,290 sec of execution time. Compared to these numbers in the FullMesh algorithm, the y-axis in

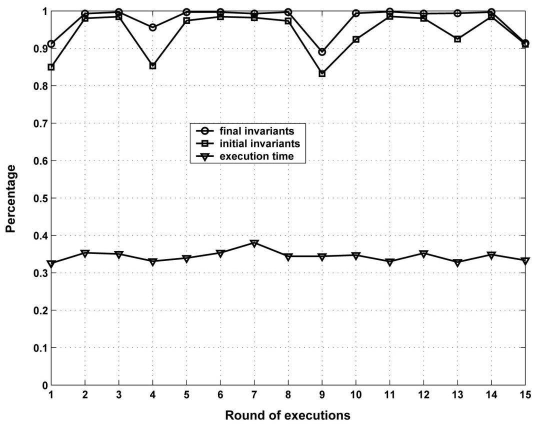


Fig. 13. Comparison between the SmartMesh and FullMesh algorithms.

Fig. 13 refers to the percentage of invariants extracted in the SmartMesh algorithm or the percentage of execution time used in the SmartMesh algorithm. Note that in this paper, we use the 1,158 initial invariants and 975 final invariants extracted from the FullMesh algorithm as the ground truth to evaluate the SmartMesh and SimpleTree algorithms. As mentioned earlier, the percentage of initial invariants is not important if we keep a high percentage of final invariants in the SmartMesh algorithm, as shown in Fig. 13. In Table 1 and Fig. 13, we observe that on the average, the SmartMesh algorithm only uses 1/3 of the execution time used in the FullMesh algorithm but extracts over 97 percent of invariants resulting from the FullMesh algorithm. Recall and precision are the common metrics used to evaluate the accuracy of document search in information retrieval [3]. Here, we borrow these two metrics to evaluate the accuracy of invariant search. The average recall of the final invariant search in the SmartMesh algorithm is 97 percent, whereas its precision is always 100 percent because all invariants extracted by the SmartMesh algorithm are always included in the set of invariants extracted by the FullMesh algorithm. Note that the average recall of initial invariant search in the SmartMesh algorithm is 94 percent. In fact, the curves of initial invariants and final invariants shown in Fig. 13

TABLE 2
Results of the SimpleTree Algorithm

No.	initial invariants	final invariants	time(s)
1	1069 (1057)	907 (892)	241
2	1146 (1109)	943 (910)	234
3	1121 (1094)	933 (913)	197
4	1163 (1124)	949 (939)	265
5	999 (973)	781 (780)	200
6	1176 (1131)	1017 (957)	201
7	923 (892)	904 (846)	233
8	1157 (1125)	998 (960)	230
9	1163 (1124)	987 (947)	227
10	1070 (1061)	912 (896)	223
11	1160 (1126)	1005 (954)	204
12	996 (975)	789 (789)	232
13	1164 (1124)	990 (952)	215
14	1174 (1127)	1013 (953)	214
15	1172 (1131)	1039 (967)	215
avg.	**1110 (1078)**	**944 (910)**	**222**

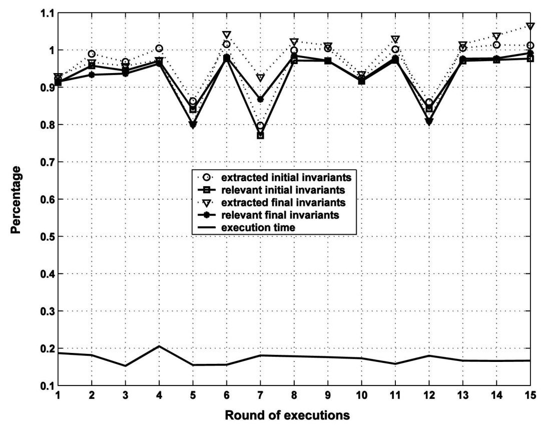


Fig. 14. Comparison between the SimpleTree and FullMesh algorithms.

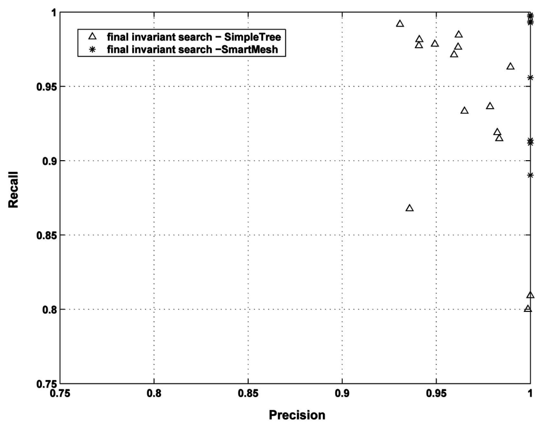


Fig. 15. Recall and precision in invariant search.

exactly represent the recall of the SmartMesh algorithm at each round, whereas their precision is always 100 percent.

The SimpleTree algorithm is also applied on the same monitoring data to search invariants. We run the algorithm 15 rounds, and the results are shown in Table 2. Each column has the same meaning as in Table 1. The initial invariants are the invariants extracted by the SimpleTree algorithm, and the final invariants are the portion of initial invariants that finally pass the same seven sequential testing phases as mentioned above. As discussed earlier, all invariants extracted by the SmartMesh algorithm are always included in the set of invariants extracted by the FullMesh algorithm so that the number of invariants extracted by the SmartMesh algorithm will never be larger than that extracted by the FullMesh algorithm. However, in the SimpleTree algorithm, we derive many models using (23)-(26) rather than learn these models using real monitoring data and (7). As shown in Table 2, for some rounds, the SimpleTree algorithm extracts more invariants than the FullMesh algorithm does. This is because the order of derived models is always larger than the order used to learn models so that the derived models may overfit the monitoring data. As discussed in Section 8, in a triangle relationship, if two learned models have orders such as $[n, m]$ and $[\hat{n}, \hat{m}]$, respectively, the order of their derived model will be $[n + \hat{n}, m + \hat{m}]$, which is much larger than the orders of learned models. As mentioned earlier, we set $0 \leq n$ and $m \leq 2$ to learn models in our experiments. As the order of model structure $[n, m]$ in (1) increases, the fitness score monotonically increases too, and we could learn models that overfit the monitoring data. The increasing flexibility of the model structure eventually enables the model to fit noise well (that is, overfit); however, noise changes randomly over time and does not reflect real flow intensities.

In Table 2, the numbers in parentheses refer to how many invariants extracted by the SimpleTree algorithm are also included in the set of invariants extracted by the FullMesh algorithm. For example, at the first round of execution, among the 1,069 initial invariants extracted by the SimpleTree algorithm, 1,057 invariants are also included in the set of 1,158 initial invariants extracted by the FullMesh algorithm. Meantime, among the 907 final invariants left after the sequential testing phases, 892 invariants are included in the set of 975 final invariants extracted by the FullMesh algorithm. We manually check the new added invariants in the SimpleTree algorithm and verify that these new invariants are added because of data overfitting. All added invariants are associated with several noisy flow intensity measurements that are unlikely to have strong relationships with other measurements. The following sequential testing process filters out some of these overfitted models, but the seven testing phases used in our experiments seem not to be sufficient for filtering out all of them. According to Table 2, on the average, over 96 percent of both initial and final invariants extracted by the SimpleTree algorithm are also included in the set of invariants extracted by the FullMesh algorithm, that is, the average precision of the SimpleTree algorithm is 96 percent. Meantime, it is easy to conclude that the average recall of both initial invariant search and final invariant search in the SimpleTree algorithm is 93 percent. With a little loss in accuracy, the SimpleTree algorithm on the average only takes 1/6 of the execution time used in the FullMesh algorithm to search invariants. In our experiments, the SimpleTree algorithm reduces computational time from 1,290 sec used in the FullMesh algorithm to 222 sec.

Fig. 14 illustrates the performance comparison between the SimpleTree algorithm and the FullMesh algorithm. The curves of "initial invariants" and "final invariants" have the same meaning as in Fig. 13. In this figure, we use "extracted invariants" and "relevant invariants" to distinguish the total number of invariants extracted by the SimpleTree algorithm and the number of relevant invariants shown in the parentheses in Table 2. Compared to the SmartMesh algorithm, the SimpleTree algorithm can further reduce computational time but is less accurate in invariant search. The SimpleTree algorithm may derive false invariants because of data overfitting, and its precision is always lower than that of the SmartMesh algorithm (100 percent). Fig. 15 illustrates the recall and precision results of the final invariant search in the two algorithms. Note that the search accuracy of initial invariants is not as important as that of final invariants because only final invariants are used to characterize systems and support system management tasks. In practice, we should consider both efficiency and accuracy requirements from a specific problem to select these algorithms.

The accuracy of both SmartMesh and SimpleTree algorithms is much dependent on the level of measurement noises and modeling errors ϵ. For example, assuming that

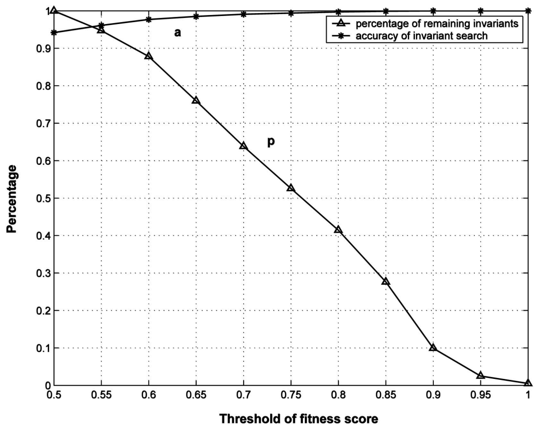


Fig. 16. Accuracy and invariant numbers versus threshold of fitness scores.

there is no modeling error for each model (that is, $\epsilon = 0$), then both algorithms should find as many invariants as the FullMesh algorithm does and get 100 percent accuracy in invariant search. As discussed before, we could have such a conclusion because both algorithms use the transition property of linear functions to reduce search complexity. Without modeling errors, such a transition property always holds in a triangle relationship of invariants so that our new algorithms will not lose any invariants while speeding up.

In this paper, we use the threshold of fitness scores $\widetilde{F}$ to control the level of modeling errors ϵ. If we raise the threshold of fitness scores, we extract a smaller number of invariants, but all extracted invariants should have smaller modeling errors ϵ. Our algorithms will get better accuracy if the threshold $\widetilde{F}$ becomes higher. As discussed earlier, this is because the transition property of linear functions is more likely to hold under smaller modeling errors. As an example, Fig. 16 shows how the accuracy of the SmartMesh algorithm is improved with the increase of thresholds. For each threshold, we run experiments five times to get the average accuracy, which is denoted by a here. As discussed before, the FullMesh algorithm extracted total 1,158 initial invariants at threshold $\widetilde{F} = 50$. At different thresholds, the number of invariants extracted by the FullMesh algorithm is divided by this 1,158 to get the percentage of remaining invariants, which is denoted by p here. Fig. 16 also shows how this percentage p decreases as we raise the threshold. At each threshold, the number of invariants extracted by the FullMesh algorithm can be derived by $1,158 * p$, whereas the number of invariants extracted by the SmartMesh algorithm can be further derived by $1,158 * p * a$.

Therefore, due to measurement noises and modeling errors in engineering practice, theoretically, any reduction of search complexity from T_1 may potentially lead to some loss of invariants. We have to trade off accuracy with computational time, and there are no other alternatives. Since we usually extract a large number of invariants from large systems, the set of invariants includes much redundancy in profiling systems. For example, we may only need two models rather than three in a triangle relationship for system characterization. In practice, we may not need the complete set of invariants for system management tasks most of the time, and our approximate algorithms are then useful to reduce the computational complexity.

10 Conclusions

Large-scale distributed systems such as Internet services could consist of thousands of components including servers, software, networking devices, and storage equipments. These system components produce a large amount of monitoring data to track their operational status. However, it is hard to effectively correlate such monitoring data for operational system management due to the difficulty to precisely characterize large and complex distributed systems. We measure flow intensities at various points across distributed systems and systematically search invariant relationships among flow intensity measurements. If a relationship holds all the time, we regard it as an invariant of the underlying system. Such invariants can be used to characterize complex systems and support various system management tasks.

Our previous invariant search algorithm has high computational complexity, and it may not scale well in large systems and networks. To this end, we proposed two efficient but approximate algorithms to search invariants from monitoring data. Both randomized algorithms could significantly reduce the computational complexity in invariant search while achieving good approximation accuracy. Experimental results from a real system are also included to demonstrate the efficiency and accuracy of our new algorithms. In our future work, we will continue to investigate whether we can derive a tight bound of computational complexity for our new algorithms and further verify their efficiency and accuracy in large systems.

References

[1] G. Adomavicius and A. Tuzhilin, "Using Data Mining Methods to Build Customer Profiles," *Computer,* vol. 34, no. 2, pp. 74-82, 2001.
[2] M.F. Arlitt and C.L. Williamson, "Web Server Workload Characterization: The Search for Invariants," *ACM SIGMETRICS Performance Evaluation Rev.,* vol. 24, no. 1, pp. 126-137, 1996.
[3] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval,* first ed. Addison-Wesley, 1999.
[4] W. Brogan, *Modern Control Theory,* third ed. Prentice Hall, 1990.
[5] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, Indexing, Clustering, and Retrieving System History," *SIGOPS Operating Systems Rev.,* vol. 39, no. 5, pp. 105-118, 2005.
[6] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms,* first ed. MIT Press and McGraw-Hill, 1990.
[7] M. DeGroot and M. Schervish, *Probability and Statistics,* third ed. Addison-Wesley, 2001.
[8] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *IEEE Trans. Software Eng.,* vol. 27, no. 2, pp. 99-123, Feb. 2001.
[9] J. Gertler, *Fault Detection and Diagnosis in Engineering Systems.* Marcel Dekker, 1998.
[10] S. Hangal and M. Lam, "Tracking Down Software Bugs Using Automatic Anomaly Detection," *Proc. 24th Int'l Conf. Software Eng. (ICSE '02),* pp. 291-301, 2002.
[11] J. Hoxmeier and C. DiCesare, "System Response Time and User Satisfaction: An Experimental Study of Browser-Based Applications," *Proc. Sixth Americas Conf. Information Systems (AMCIS '00),* pp. 140-145, 2000.
[12] R. Isermann and P. Balle, "Trends in the Application of Model-Based Fault Detection and Diagnosis of Industrial Process," *Control Eng. Practice,* vol. 5, no. 5, pp. 709-719, 1997.

[13] G. Jiang, H. Chen, and K. Yoshihira, "Discovering Likely Invariants of Distributed Transaction Systems for Autonomic System Management," *Proc. Third Int'l Conf. Autonomic Computing (ICAC '06),* pp. 199-208, June 2006.

[14] G. Jiang, H. Chen, and K. Yoshihira, "Modeling and Tracking of Transaction Flow Dynamics for Fault Detection in Complex Systems," *IEEE Trans. Dependable and Secure Computing,* vol. 3, no. 4, pp. 312-326, Oct.-Dec. 2006.

[15] N. Jiang, R. Villafane, K. Hua, A. Sawant, and K. Prabkakara, "ADMiRe: An Algebraic Data Mining Approach to System Performance Analysis," *IEEE Trans. Knowledge and Data Eng.,* vol. 17, no. 7, pp. 888-901, Aug. 2005.

[16] L. Ljung, *System Identification—Theory for The User,* second ed. Prentice Hall, 1998.

[17] D. Menasce, V. Almeida, R. Riedi, F. Ribeiro, R. Fonseca, and W. Meira, "In Search of Invariants for E-Business Workloads," *Proc. Second ACM Conf. Electronic Commerce (EC '00),* pp. 56-65, 2000.

[18] R. Motwani and P. Raghavan, *Randomized Algorithms.* Cambridge Univ. Press, 1995.

[19] J. O'Madadhain, D. Fisher, S. White, and Y. Boey, "The Jung (Java Universal Network/Graph) Framework," Technical Report UCI-ICS 03-17, UC Irvine, Dept. of Information and Computer Science, jung.sourceforge.net, 2003.

[20] D. Oppenheimer, A. Ganapathi, and D. Patterson, "Why Do Internet Services Fail, and What Can Be Done about It," *Proc. Fourth Usenix Symp. Internet Technologies and Systems (USITS '03),* pp. 1-16, 2003.

[21] D. Patterson and A. Brown et al., "Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies," Technical Report UCB//CSD-02-1175, UC Berkeley, Dept. of Computer Science, roc.cs.berkley.edu, 2002.

[22] J. Perkins and M. Ernst, "Efficient Incremental Algorithms for Dynamic Detection of Likely Invariants," *Proc. ACM 12th Symp. Foundations of Software Eng. (FSE '04),* pp. 23-32, Nov. 2004.

[23] http://java.sun.com/developer/releases/petstore/, 2006.

[24] J.O. Smith, *Math. of the Discrete Fourier Transform (DFT).* W3K Publishing, 2003.

[25] M. Spiliopoulou, C. Pohle, and L. Faulstich, "Improving the Effectiveness of a Web Site with Web Usage Mining," *Proc. Int'l Workshop Web Usage Analysis and User Profiling (WEBKDD '99),* pp. 142-162, 2000.

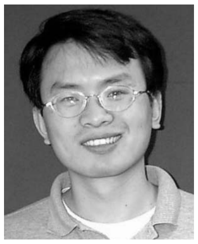
[26] J. Srivastava, R. Cooley, M. Deshpande, and P. Tan, "Web Usage Mining: Discovery and Applications of Usage Patterns from Web Data," *ACM SIGKDD Explorations Newsletter,* vol. 1, no. 2, pp. 12-23, 2000.

[27] L. Wolsey and G. Nemhauser, *Integer and Combinatorial Optimization.* Wiley-Interscience, 1999.

[28] Q. Yang, H. Zhang, and T. Li, "Mining Web Logs for Prediction Models in WWW Caching and Prefetching," *Proc. Seventh ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining (KDD '01),* pp. 473-478, 2001.

[29] O. Zaiane, M. Xin, and J. Han, "Discovering Web Access Patterns and Trends by Applying Olap and Data Mining Technology on Web Logs," *Proc. IEEE Forum on Research and Technology Advances in Digital Libraries (ADL '98),* pp. 19-29, Apr. 1998.

[30] G. Zhen, G. Jiang, H. Chen, and K. Yoshihira, "Tracking Probabilistic Correlation of Monitoring Data for Fault Detection in Complex Systems," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '06),* pp. 259-268, June 2006.



Guofei Jiang received the BS and PhD degrees in electrical and computer engineering from the Beijing Institute of Technology, China, in 1993 and 1998, respectively. During 1998-2000, he was a postdoctoral fellow in computer engineering at Dartmouth College, New Hampshire. He is currently a senior research staff member in the Robust and Secure Systems Group, NEC Laboratories America, Princeton, New Jersey. His current research focus is on distributed system, dependable and secure computing, and system and information theory. He has published nearly 50 technical papers in these areas. He is an associate editor of the *IEEE Security and Privacy* magazine and has severed in the program committees of many prestigious conferences.

Haifeng Chen received the BEng and MEng degrees in automation from Southeast University, China, in 1994 and 1997, respectively, and the PhD degree in computer engineering from Rutgers University, New Jersey, in 2004. He has worked as a researcher in the Chinese National Research Institute of Power Automation. He is currently a research staff member at NEC Laboratories America, Princeton, New Jersey. His research interests include data mining, autonomic computing, pattern recognition, and robust statistics.

Kenji Yoshihira received the BE degree in electrical engineering from the University of Tokyo in 1996 and the MS degree in computer science from New York University in 2004. He designed processor chips for enterprise computer at Hitachi Ltd. for five years. He was the chief technical officer (CTO) at Investoria Inc., Japan, developing an Internet service system for financial information distribution through 2002. He is currently a research staff member with the Robust and Secure Systems Group, NEC Laboratories America, New Jersey. His current research focus is on distributed system and autonomic computing.