

Autonomous Hot Patching for Web-Based Applications*

Hai Huang Wei-Tek Tsai Yinong Chen
Department of Computer Science and Engineering
Arizona State University Tempe, AZ 85281, U.S.A.

{hai, wtsai, yinong}@asu.edu Phone: +1-480-727-6921 Fax: +1-480-965-2751

Abstract

Patching technologies are commonly applied to improve the dependability of software after release. This paper proposes an autonomous hot patching (AHP) framework to fully automate the reasoning for the causes of failures, and to patch the binary code of Web-based applications. AHP admits the hardness for rooting out all faults before product release, and autonomously patches problems of application programs. By directly operating on binary code, AHP is universal to virtually all applications. A promising application of AHP is to shortcut the remote maintenance center (RMC) and hence to reduce the turn around time for patches.

1 Introduction

It is difficult to eliminate all faults before releasing software thus patches are issued to repair faults during operation. Frequently experiencing crashes and applying patches are annoying to end users. One theme of patching technologies is to improve the user experience, e.g., the “hot patching” [8] that aims to reduce the number of mandatory system reboots. When a failure occurs in an application, relevant data will be collected and sent to the Remote Maintenance Center (RMC) where the data are analyzed, faults are identified, and a patch is issued. The patch will then be downloaded and applied to the system. If a patch must be generated manually, users may wait for days, which is not desirable. It also slows down the collection of timely runtime data from the RMC.

A Web-based application consists of Web Services [12, 11, 10] as its components. This paper propose an *autonomous hot patching* (AHP) framework that allows systems to learn the possible causes of failures and to patch the binary code of Web-based applications (referred to as application hereafter). AHP improves user experience by short-

cutting the RMC and reducing turn around time for patches. The overall process of AHP is as follows. When an application crashes, AHP will be notified. From the crash report generated by the OS, AHP determines the subroutine or the Web method (referred to as method hereafter) that witnesses the crash. Via binary level data flow analysis and dependency analysis, AHP assumes that all the local variables of the method and global variables the method depends on are the potential causes of the crash. The assumption is feasible if the causes of the failure reside in the application, instead of the OS or other applications. AHP then instruments the binary code of the application to monitor those variables. The data will be fed into a Boolean function learner to reason the causes of the crash. Note that data for normal execution contribute to reasoning as well, because they tell what do not lead to the crash. Once the causes are identified, AHP will patch the binary code to automatically circumvent a potential crash when the causes occur again. In addition, the user may have choices to retry some operations, or to correct some problematic data to avoid the crash.

The key advantages of AHP are its *full automation* and *operating on binary code*. Full automation allows zero dependency on RMC and hence exploits the proximity to applications. AHP is applicable to virtually all applications by directly operating on binary code. It also reduces the hardness of reasoning due to the tenuous semantics in the binary code.

AHP is not aimed to replace RMC. The patches issued by AHP can be viewed as the “first aid” that circumvents undesirable behaviors of application such as crashes. Thus the user can continue their current work and wait for the canonical patch issued from RMC. AHP can also collaborate with RMC, such as to collect more runtime data and to apply patches on behalf of RMC. The techniques developed for automating AHP would as well facilitate RMC to accelerate the process of pinning down faults and generating generic patches.

AHP is different from self-healing systems [3, 4]. Self-healing systems fall into two broad categories: *Fault-tolerant systems* that uses a variety of techniques, such as re-

*The research is partially supported by Microsoft Research Phoenix Project 2005, and NSFITR, NISTP, and CEINT.

dundancy and graceful degradation, to react properly upon failures; and *environment-aware systems*, also known as *resource awareness* or *situation awareness*, which react to environment variation and adapt themselves to function optimally.

Though AHP offers some “self-healing” capability by autonomously patching the applications running on the system, it deviates from either category of self-healing systems. AHP backs off from the ambition to predicate every possibility a priori and prepares for the unpredictable or hardly predictable problems. AHP does not require redundant components; it instruments and patches existing components. In AHP, there exists two systems, the OS and the applications. Hence, it avoids the problem that a faulty system heals itself without incurring new faults.

Instrumentation techniques are widely applied in self-healing systems, such as detecting faults, repairing failures [2, 3], and enforcing policies [6]. In a general instrumentation framework [7], a instrumentation “recipe” is required to indicate what to monitor. AHP does not require such a “recipe” as it learns what to monitor autonomously.

AHP incorporates three algorithmic techniques: data mining, (probabilistic) Boolean function learning, and model checking. Data mining abstracts data (variable, value pairs) to be predicates (Boolean variable, truth value pairs) and reduces the reasoning to be a Boolean function learning problem. Probability is introduced to Boolean function learning to cope with the uncertainty introduced by data mining or the natural nondeterminism of the causes. Model checking techniques are applied to back track the state transitions and compute a *remote cause*. Remote causes are the causal states long before the crash. Remote causes are desirable because it might be too late to correct the problem right before the crash. For example, if the crash is in the middle of a transaction, it is hard for AHP to roll back properly.

The rest of the paper is organized as follows. Section 2 elaborates the AHP framework and discusses how to apply AHP to embedded systems and how AHP collaborate with RMC. Section 3 presents the Boolean function learning algorithm. Section 4 employs an experiment to illustrate how AHP works. Section 5 concludes the paper.

2 Autonomous Hot Patching

2.1 AHP framework

The AHP operations are divided into three phases: data collection, reasoning, and patching. Data collection is responsible for determining what variables to monitor and to instrument the binary code of the applications. Reasoning phase constructs the boolean functions that represents the causes of the failures from the collected data. Patching phase selects the patch point and changes the binary code

accordingly to circumvent the faults. Figure 1 illustrates the framework.

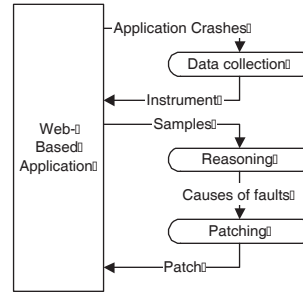


Figure 1. AHP framework

Data collection When an application crashes, the OS often generates a crash report recording the environment at the crash point, such as the error report by Windows systems or the core dump by Unix / Linux systems. The crash reports usually consists of a full memory image and hence it is tedious to parse it. Instead of trying to figure out the cause of the crash from the crash report, treats it as a trigger to start data collection on the crashed application. One type of information in the crash report helpful for the data collection is the method that witnesses the crash. Since the crash occurs within the method, it is reasonable to assume that some combination of the data referenced by the method causes the crash.

Let S denote a method, x denote a variable, v denote a value, and V denote a set of values. A pair (x, V) of a variable x and value set V means that x holds a value $v \in V$. Denote (x, V) as an *atomic cause*. For example, a common cause of memory violation is to read from a null pointer, which is expressed as an atomic cause $(ptr, \{null\})$. An atomic cause can be viewed as a predicate and the abstraction simplifies the reasoning phase to be Boolean function learning only. A *complex cause* is a set of atomic cause connected by Boolean operators \vee , \wedge , and \neg . A cause is *deterministic* if its presence always leads to crash. Otherwise it is *nondeterministic* or *probabilistic*. A cause is *local* if all variables in the cause are local to the application.

This paper considers deterministic local causes. Thus only the parameters (denoted by L) of S and the global variables (denoted by G) that S depends on form the cause. It is sufficient to monitor variables in $L \cup G$. L and G can be computed by data flow analyses and dependency analyses on the binary code of S , where variables are locations in the stack, heap, or the data segment. Phoenix facilitates data flow analyses and dependency analyses.

To reduce the overhead, we insert the data collection at the beginning of S to record the values of all variables in $L \cup G$. We also record whether S crashes or not. Both types

of information are fed into the reasoning phase.

Phoenix provides powerful functions to manipulate the binary code. It can traverse the binary code and insert instructions. It even allows insertion of a call to a method located in another dynamic linkable library (DLL). To accommodate data collection, we prepare a DLL exporting recording functions for different data types: byte, short integer, and long integer. Then we traverse the binary code, locate the entry point of S , insert calls to proper recording functions to record the values.

Note that the data collection can only collect variable-value pairs (x, v) , instead of the variable-value set pairs (x, V) required by the reasoning phase. Data mining is introduced to cluster values together to form the proper value sets according to whether S crashes or not. Specific values to some type variables, such as null value to pointers, are of particular interesting and can be predefined as value sets. The conditions that guards the branching in the application also provide hints for constructing value sets. For example, if the program checks whether $salary \geq 1000$, then $(salary, [1000, +\infty))$ and $(salary, (-\infty, 1000))$ are good candidates.

Data mining may introduce uncertainty and the reasoning phase is extended to handle probabilistic causes to cope with the uncertainty.

Reasoning An atomic cause is a predicate. A complex cause is a Boolean expression, or a Boolean function $f : \Omega = \{0, 1\}^n \rightarrow \{0, 1\}$, where $n = |L \cup G|$. Define that $f(\omega) = 1$ denotes $\omega \in \Omega$ causes the crash. Each record collected by the data collection is a *sample* $(\omega, f(\omega))$ of the Boolean function. A *positive sample* is a sample where $f(\omega) = 1$, and a *negative sample* is where $f(\omega) = 0$. Positive samples causes crashes. The reasoning phase is to learn the Boolean function from a set of samples. Section 3 details the Boolean function learning algorithm.

Patching After determining the cause, we can patch the binary code to monitor whether the cause occurs and to circumvent the fault if any. A *patch point* is a location in the program where one apply the patch. To reduce the patching overhead, we want to reduce the number of patch points. A straightforward selection is the entry point of S , where one collects data. Recall that in some cases a remote cause might be a better selection as a patch point. Model checking techniques [1] are applied to compute remote patch points.

The way the patch works affects the selection of patch points. Once a cause occurs, the patch can *retry* some operations (to *modify* some data), *abort* the program, or *ignore* the cause. Retry is to roll back to an early location in the program and hopefully the re-execution of the operations will resolve the cause. For example, the user can re-input the problematic data, or redo the problematic operation; or the program can redo some I/O operations to retrieve normal data from the communication links to other applica-

tions or systems. We propose three types of *retry points*: entry points of methods, I/O points, and entry points of critical regions. Retry a method allows a complete re-run of a method. I/O points are where the program read or write data. For example, the user inputs data through console or popup dialogs, or data are read from or written to a socket, pipe, or file. Retry at I/O points offers opportunities to correct error data input or output. Usually critical regions are used for transactions or exception handling. Retry at the entry point of critical regions preserves the integrity of the transaction or the exception handling, and removes from AHP the burden for rollback. Abort terminates the application before further damages are incurred. Ignore simply does nothing and continues the program.

Retry points are categorized into two classes: *idempotent* retry points and *nonidempotent* retry points. A retry point is idempotent if every re-execution of it produces identical results. Otherwise, it is nonidempotent. Examples for idempotent retry point are user's inputs. Examples for non-idempotent retry points are I/O operations on a stream because each I/O operation moves the current pointer to the stream. If the retry point is nonidempotent, then additional cares are needed to roll back to corresponding states before retry. In practice, it is usually hard to perform rollback without setting up checkpoint before retrying. Hence we retry only at idempotent points. Data flow analyses and dependency analyses are applied to determine the idempotency. We define a *precedence* relation among instructions in binary code. Instruction I_i precedes I_j if there exists an execution path that is shared by I_i and I_j , and I_i is executed before I_j on that path. A retry point r is idempotent if for all locations preceded by r , their states are uniquely determined by the states at r . Due to the existence of loops, it could happen that I_i precedes I_j and I_j precedes I_i , which does not affect the consistency of the above definition on idempotent.

Patch point and retry point can differ from one another. That is, one can monitor the cause at a later point (patch point) and then retry the operation at an early point (retry point), provided the retry point is idempotent. Patch point is determined by the cost for monitoring the cause. The *complexity* of a cause is the number of sufficient comparisons to determine the cause. The *complexity* of an atomic cause (x, V) is the number of sufficient comparisons to verify whether $x \in V$. If V is a discrete value set, then the complexity of (x, V) is upper bounded by $|V|$. If V is a continuous value set, $V = \bigcup_{i=1}^n W_i$, where W_i are intervals, then the complexity of (x, V) is upper bounded by $2n$. The complexity of a cause is then upper bounded by the sum of the complexity of all its atomic causes. Due to the state transition along execution, it can happen that at the retry point the cause is far more complex. In this case we can reduce the complexity by choosing a patch point different

from the retry point. Model checking techniques and data-flow analysis techniques can be applied to back-track to a point before the retry point which may incur less patching complexity.

If one cannot find a proper retry point, one can modify the values of some variables in the cause to resolve the faults. It is desirable that an engineer is consulted to determine the modification. However, since only binary code is available, it is hard for the engineer to interpret the meanings of variables. Variable names can be retrieved if a debug environment is available, e.g., the *.PDB file generated by Phoenix. If an engineer is not available, AI can be applied to perform the modification according to historical records. The data collection records all negative and positive samples and it is feasible to modify the cause to a proximate negative sample. For variable-value pairs (x, v_i) and (x, v_j) , the *distance* between them is defined as $|v_i - v_j|$. The distance between two samples is defined as the accumulated distances of all variable-value pairs. Multiple choices are available to define the accumulated distance, such as 1-norm, 2-norm, or ∞ -norm. The patch logs the modification for later review.

Figure 2 summarizes the structure of the patch. Users can customize the patch by embedding their own handling function within the structure.

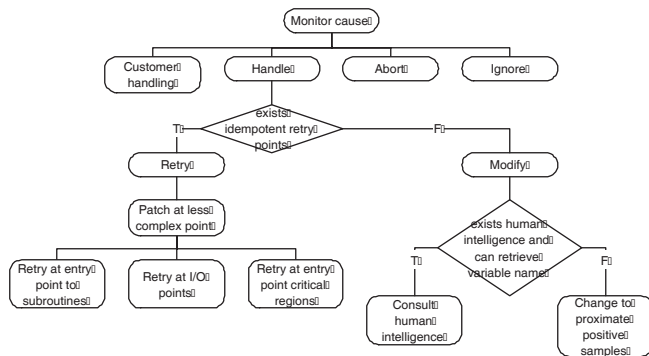


Figure 2. Structure of the patch

Patch is applied via Phoenix. The monitoring and the handling parts are translated to binary instructions and inserted into the binary code at proper location.

2.2 Supporting Techniques

Phoenix A key enabling technique is the Phoenix project from Microsoft [9]. Phoenix is a framework for building various extensions of compilers. Phoenix involves advanced techniques in compilation and program analysis:

- Basic Block Analysis
- Memory Tracing

- Code Coverage
- Fault Injection
- Run-time Profiling and Feedback
- Ahead-of-Time and Just-in-Time Compilers
- Whole-program, Post-link, and Runtime Optimization

Phoenix generates code for a wide range of processor architectures. AHP exploits the binary analysis and manipulation capability to treat binary code of applications. See <http://research.microsoft.com/phoenix/> for more information of Phoenix.

Embedded systems Special cares are necessary due to the limited energy and computation capacity of embedded systems. We propose the “device + host” model to cope with the problem of limited resources. The device only has the data collection capability. The host PC will reason the causes and generate the patch, which will be deployed automatically to the device when it is connected to the host PC, e.g., when the Pocket PC is sited into its cradle. One can further reduce the overhead by removing the data from the device upon each connection. We applied AHP to develop a demo system consisting of a robot car, a Pocket PC (WinCE), and RMC, see the demo at <http://asur1.eas.asu.edu/EmbeddedExplorer/>.

Collaboration with RMC The runtime data collected by AHP can be sent to facilitate RMC. AHP can apply the patches on behalf of RMC. RMC can adopt the techniques developed for AHP to pin down possible causes of the fault, such as binary code analyses, binary code manipulation, Boolean function learning, data mining, and model checking techniques.

Evaluation We propose three quantitative metrics to evaluate the effectiveness and efficiency of AHP: convergency, overhead, and accuracy. *Convergency* indicates how fast the Boolean function learning determines the cause of the crash. We propose to measure the convergency by the number of samples collected for the reasoning phase to be effective. Define a *positive* sample to be a sample that causes the crash and a *negative* sample to be the opposite. Two sub-metrics are proposed, the positive convergency — the number of positive samples, and the negative convergency — the number of negative samples. Because negative samples do not cause crashes, user experience is mainly affected by positive convergency. *Overhead* measures how many instructions are inserted to the binary code of application, which bounds the complexity. There are two types of overhead: data collection overhead and patch overhead, which are the number of binary instructions for data collection and patch, respectively. *Accuracy* is the probability that the cause learnt by the reasoning phase is correct. The *effectiveness* of AHP is measured by the combination of fast convergency, high accuracy, and less overhead.

3 Boolean Function Learning

Boolean function learning is a popular technique topic [5, 13]. The problem setting of AHP does not exactly fit with the classical one due to the asymmetry between positive and negative samples. It is desirable to have few positive samples to improve user experiences. Much more negative samples are affordable.

We formulate new criteria for Boolean function learning. Without loss of generality, assume Boolean functions are expressed in disjunctive normal form (DNF). A k -DNF is a DNF such that each conjunct consists of at most k literals. Note that k reflects the complexity of the DNF. Let f and g be two Boolean functions. We say that f *overapproximates* g if $f \rightarrow g$ is a tautology. For example, $A \wedge B$ overapproximates A , which in turn overapproximates $A \vee B$. We abuse the notation \rightarrow to also denote the overapproximation relation, i.e., $f \rightarrow g$ denotes f overapproximates g . Clearly overapproximation is transitive and reflexive. If $f \rightarrow g$, then $f(\omega) = 1$ implies $g(\omega) = 1$, and $g(\omega) = 0$ implies $f(\omega) = 0$. The deterministic Boolean function learning problem is to construct a DNF f from a set of samples $\Omega = \{\omega_1, \dots, \omega_n\}$, such that $\forall \omega_i, \omega_i$ is a positive sample, $\omega_i \rightarrow f$, and $\forall \omega_j, \omega_j$ is a negative sample, $f \rightarrow \neg \omega_j$, where $\Omega = \{0, 1\}^m$ is a m -dimensional Boolean vector space. Suppose that f is a k -DNF, we want k to be as small as possible to reduce the complexity. Usually, we expect much smaller k than n , because software developers tend to use simple conditions.

Establish a reverse lattice as shown in Figure 3. The top, the 0-th level, is a singleton \perp . The k -th level consists of all possible conjuncts of k literals. Then connect an entry f in the k -th level to an entry g in the $(k-1)$ -th level if $f \rightarrow g$. We define that $p \rightarrow \perp$ for all literals p . Algorithm 1 first constructs an overapproximation f of all positive samples using entries with as lower level as possible in the reverse lattice. Then it examines all negative samples against f and push to higher levels in the reverse lattice when necessary.

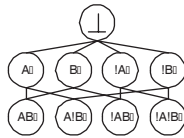


Figure 3. Reverse lattice

Theorem 1. *Algorithm 1 returns an overapproximation f such that $\forall \omega_i, \omega_i$ is a positive sample, $\omega_i \rightarrow f$, and $\forall \omega_j, \omega_j$ is a negative sample, $f \rightarrow \neg \omega_j$. The runtime is $O(\sum_{j=1}^k \binom{m}{j} n)$, where n is the number of samples, m is the dimension of Ω , and k is the maximum length of conjunct in the resulting DNF.*

Algorithm 1: Deterministic Boolean function learning

- 1: Determine $k \leq n$, construct the reverse lattice to the k -th level, mark all entries f in the lattice to be *black*
 - 2: **foreach** positive sample ω_i **do**
 - 3: **foreach** f in the reverse lattice **do**
 - 4: Mark f to be *white* if $\omega_i \rightarrow f$
 - 5: **end**
 - 6: **end**
 - 7: **foreach** negative sample ω_j **do**
 - 8: **foreach** *white* f in the reverse lattice **do**
 - 9: Mark f to be *black* if $\neg(f \rightarrow \neg \omega_j)$
 - 10: **end**
 - 11: **end**
 - 12: **return** concise DNF determined by all *white* f
-

A probabilistic extension of the Boolean function learning algorithm is available. We omit it due to space limitation.

4 An Experiment

An experiment is developed to demonstrate the feasibility of AHP. Simplifications are taken to reduce the complexity. A Web-based application, ATM32.EXE is developed to simulate ATM activities. The program invokes four Web methods to input account number, current date, the amount, and whether to save or withdraw. The input will be fed into a method: *process*, which simulates the processing on the request and prints out information accordingly.

A fault is injected into the program to write to a null pointer when the amount is invalid, which will crash the application. Upon crashes, AHP is triggered and the analysis of the crash report indicates that crash occurs within method *process*. Further data flow analysis and dependency analysis indicate that the *process* only depends on its four parameters. Then the binary code of ATM32.EXE is instrumented for data collection. The instrumented application is stored as another program, ATM32I.EXE, which will record the values of the four parameters to a log file. Another application REASONER.EXE is developed to monitor the log file and reason the cause.

After several executions of ATM32I.EXE, REASONER.EXE figures out that the third parameter, the amount, equals to 0 leads to the crash. Since each input is idempotent, a patch is generated to retry the input of the amount when it equals to 0. In the experiment, we choose patch point to be equal to retry point to reduce complexity. The patched application, ATM32P.EXE will popup a dialog asking whether to retry, abort, or ignore when it sees that the third input is 0. If the user chooses retry, it jumps back

to the third method input and allows the user to re-input. The user can input 1 to avoid the crash.

Figure 4 shows the binary code before and after the patch. Code in italic is the patch code. Label REDO: marks the redo point. Instruction `call __imp_Patch@0, {ESP}` pops up a dialog and returns the users choice.

```

{ESP}      = call &InputAccount, {ESP}
[EBP-4]    = mov AL
{ESP}      = call &InputDate, {ESP}
[EBP-1]    = mov AL
REDO: (refs=1)
{ESP}      = call &InputAmount, {ESP}
[EBP-2]    = mov AL
EAX      = movzx [EBP-2]
EFLAGS   = test( Illegal) EAX, EAX
           jcc(NE) EFLAGS, L00001033, L00001024
L00001024: (refs=1)
{ESP}      = call __imp_Patch@0, {ESP}
ECX      = movzx AL
EFLAGS   = test( Illegal) ECX, ECX
           jcc(EQ) EFLAGS, L00001033, L00001031
L00001031: (refs=1)
           jmp REDO
L00001033: (refs=2)
{ESP}      = call &SaveWithdraw, {ESP}
[EBP-3]    = mov AL

```

Figure 4. Binary code before and after patch

In the experiment, it took 3 samples before REASONER.EXE reaches a decision; 2 of them are negative, and 1 is positive. The accuracy is 100% since AHP pins down the actual cause. The number of binary instructions for data collection is 6, which are basically calls to a supporting DLL to record samples. The number of binary instructions for the patch is 8, which are two comparisons, a call to supporting DLL to popup the dialog, and a jump back to the retry point. The instrumentation overhead is less than 1% in terms of numbers of instruction, and less than 5% in terms of runtime. According to the evaluation metrics, the experiment exhibits fast convergence, high accuracy, and small overhead (i.e., less intrusive).

5 Conclusion and Future Work

This paper proposes an innovative AHP framework to support autonomous patch that shortcuts the RMC and thus improve user experience by reducing significantly the turn around time for patches. An experiment is presented to illustrate the feasibility of AHP on Web-based applications. The paper also briefly discussed how AHP works for embedded systems and the collaboration between AHP and RMC.

Acknowledgement

We thank Microsoft Research, the entire Phoenix team, and especially Dr. John Lefor, for their help on issues relating to Phoenix project.

References

- [1] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2002.
- [2] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of the 18th Conference on Object Oriented Programming Systems, Languages, and Applications*, 2003.
- [3] D. Garlan, S.-W. Cheng, and B. Schmerl. Increasing system dependability through architecture-based self-repair. In *Architecting Dependable Systems*, 2003.
- [4] I. Georgiadis, J. Magee, and J. Karger. Self-organising software architectures for distributed systems. In *Proceedings of ACM SIGSOFT Workshop on Self-Healing Systems*, 2002.
- [5] M. Kearns, M. Li, L. Pitt, and L. G. Valiant. On the learnability of Boolean formulae. In *Proceedings of the 9th annual ACM Symposium on Theory of Computing*, pages 285–295, 1987.
- [6] V. Kiriansky, D. Burening, and S. Amarasinghe. Secure execution via program shepherding. In *USENIX Security Symposium*, 2002.
- [7] N. Kumar, J. Misurda, B. R. Childers, and M. L. Soffa. Instrumentation in software dynamic translators for self-managed systems. In *Proceedings of ACM SIGSOFT Workshop on Self-Managed Systems*, 2004.
- [8] Microsoft. Common engineering criteria for 2005. <http://www.microsoft.com/windowserversystem/cer/allcriteria.mspx>, 2004.
- [9] Microsoft. Phoenix project. <http://research.microsoft.com/phoenix/>, 2004.
- [10] W. Tsai, Y. Chen, R. Paul, H. Huang, X. Zhou, and X. Wei. Adaptive testing, oracle generation, and test script ranking for web services. In *Proceedings of the 29th Annual International Computer Software and Applications Conference*, 2005. To appear.
- [11] W. Tsai, W. Song, R. Paul, Z. Cao, and H. Hunag. Services-oriented dynamic reconfiguration framework for dependable distributed computing. In *Proceedings of the 28th Annual International Computer Software and Applications Conference*, pages 554–559, 2004.
- [12] W. Tsai, X. Wei, Y. Chen, B. Xiao, R. Paul, and H. Huang. Developing and assuring trustworthy web services. In *Proceedings of 7th International Symposium on Autonomous Decentralized Systems*, pages 43–50, 2005.
- [13] L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.