Error Detection Framework for Complex Software Systems

Antonio Bovenzi,
Domenico Cotroneo,
Roberto Pietrantuono
Università di Napoli Federico II
Via Claudio 21, 80125, Naples, Italy
+39 0817683874
{antonio.bovenzi, cotroneo, roberto.pietrantuono}@unina.it

Gabriella Carrozza
SESM SCARL
Via Circumvallazione Esterna di Napoli, 80014
Giugliano in Campania, Naples, Italy
+390818180650
gcarrozza@sesm.it

ABSTRACT
Software systems employed in critical scenarios are increasingly large and complex. The usage of many heterogeneous components causes complex interdependencies, and introduces sources of non-determinism, that often lead to the activation of subtle faults. Such behaviors, due to their complex triggering patterns, typically escape the testing phase. Effective on-line monitoring is the only way to detect them and to promptly react in order to avoid more serious consequences. In this paper, we propose an error detection framework to cope with software failures, which combines multiple sources of data gathered both at application-level and OS-level. The framework is evaluated through a fault injection campaign on a complex system from the Air Traffic Management (ATM) domain. Results show that the combination of several monitors is effective to detect errors in terms of false alarms, precision and recall.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging – *Error handling and recovery, monitors, diagnostics.*

General Terms
Measurement, Reliability, Experimentation

Keywords
Monitoring, Error Detection, Critical Systems

1. INTRODUCTION
In complex software systems errors occur resulting from software faults lying into the code and activated by triggering conditions that strongly depend on the propagation patterns among components and on the execution environment. The presence of residual faults i.e., of faults that escape pre-operational testing campaigns and get activated only during the system execution, often results in errors that cannot be foreseen ahead of runtime. In large and complex systems the activation of these faults causes very subtle propagation patterns that are difficult to identify, due to the complex dependencies among components, and among different layers (e.g., OS, middleware, virtual machines). This hampers the process of detection, and -then- of faults diagnosis and system recovery. A suitable way for detecting errors in these system is to built a *knowledge base* of the system runtime behavior, as much accurate as possible, to be exploited during system operational life for error detection. However, it is very hard to create this knowledge a priori, and even harder to formalize it.

To tackle these objective, we need to consider two major issues: *i)* where can we get the necessary information from, in a cost-effective manner *ii)* what kind of information do we need?

This paper presents an anomaly-based framework for error detection in critical software systems able to exploit the information available while the system is still in its development phase for gaining the necessary knowledge on the system behavior, accurately and inexpensively. Specifically, the framework relies on the idea of *i)* describing the system behavior with a combination of application-level and OS-level variables, and *ii)* of collecting their values during the *testing phase*, in order to drive the detection at runtime with the availability of a reference behavior description, obtained with minimal training overhead.

Critical systems typically foresee a quite intensive testing activity, from which the detection framework has the possibility to learn the system behavior under different perspectives. In particular, during tests, the framework learns both the desired, expected behavior of the system, which acts as reference nominal behavior, and the system failing behavior, i.e., the typical dynamics occurring when the test case executions fail. The former behavior is acquired while executing functional system testing, which generates executions representing the expected system behavior, and/or while executing other kinds of testing terminating with negative outcomes (i.e., successful executions). The latter is especially learned during the robustness and stress test campaigns, (which are usually carried out in critical systems to verify the system's ability to react to unpredictable inputs/loads), and, in general, during the execution of test cases resulting in a failure.

Behavior is described through multiple variables to capture both the communications among software modules at application-level, and dynamics occurring at OS-level. Such variables are monitored in case of the expected behavior, as well as in case of failing

EWDC '11, May 11-12, 2011, Pisa, Italy

behavior. On this base, we build the system behavioral model, which constitutes the reference to be considered at runtime. To implement the approach, we designed a framework. The framework distinguishes two phases: *i)* a *knowledge base acquisition* phase, where data about the software under test is extracted to model the correct and failing behavior of the system; *ii)* a *run-time* phase, where the acquired knowledge is used to observe the runtime behavior, and to raise alarms when it deviates from the expected one, or it is similar to an already experienced (class of) failing behavior (i.e., it is very likely that the system is going to fail). Based on the kind of alarms raised from the framework, and on the confidence score assigned to the revealed anomaly, the user infers the causes, and applies proactive actions accordingly. The framework is experimented in the context of real-world ongoing prototypes in the domain of Air Traffic Management (ATM) via a fault injection campaign. Results show that the combination of application-level and OS-level monitors is effective to detect failures in terms of precision, recall, false alarms, and with negligible impact on performance.

2. RELATED WORK

Error detection is crucial to recognize that something unexpected has occurred in the system. It can be addressed at different levels, from the hardware (e.g., machine check, division by zero, etc.), to the upmost level, i.e., the application. In any case, its aim is to detect errors before they propagate to other different component(s), and eventually lead the system to failure. Error detectors can be implemented, as local or remote modules, within software components, or at their outputs. Based on the type of interaction error detection strategies can be grouped into: c*omponent-initiated, query-based* or *log-based* techniques.

Component-initiated techniques trust on the component, which provides some information specifying if the requested services could not be satisfied [5]. For instance, exceptions can be signaled by a component when its error detection mechanisms detect an invalid service request, i.e., an interface error.

Query-based techniques rely on the presence of an external module, which is in charge of querying the monitored component, to discover latent errors [6].

Log-based techniques consist in examining the log files produced by the components, in order to figure out the system behavior by correlating different events [7].

These techniques belong to *direct detection strategies*: they try to infer the system's health by directly querying it, or by analyzing events it is able to produce (e.g., logs, exceptions). However,. before a failure occurs, the activated fault may lead the system/component to exhibit anomalous behaviors. For these reasons, ***anomaly-based detection*** approaches have been conceived for detection. They aim to compare normal behavior with the observed one, and reveal anomalous runs of the system. Thus, unlike the previous ones, these approaches, which can be labeled as *indirect detection strategies,* infer the system's health by monitoring system parameters and their interactions with the environment.

Indirect techniques have been especially adopted for intrusion detection systems. In [10] the execution of a program is characterized by tracing patterns of invoked system calls. System call traces produced during the operational phase are then compared with nominal traces to reveal intrusions. A similar approach is adopted in [9], which provides detection facilities for large-scale distributed systems running legacy code. The monitors are designed to observe the exchanged messages among the entities of the distributed system, which are then used to deduce a runtime state transition diagram executed by all the entities in the system. An anomaly-based strategy is also in [8], which exploits hardware performance counters and IPC (Inter Process Communication) signals to monitor the system behavior, and to detect possible anomalous conditions. Other solutions, such as [11], are based on statistical learning approaches. They extract recurrent execution patterns (using system calls) to model the application under nominal conditions, and to classify run-time behaviors as normal or anomalous.

Our framework falls into the category of *anomaly-based detection techniques*, because it attempts to reveal anomalous behavior at runtime with respect to the learned behavior. The framework builds on our previous work [12] on hang detection, where we adopted OS-level variables to describe the nominal behavior and detect hang failures. With respect to the described previous works, our framework relies on a combined description of the system behavior via multiple variables both at application-level and at OS-level. The aim is to monitor interactions among application components, as well as interactions between application and OS layers. A key step of the approach is also the usage of information provided during the testing phase for learning the system behavior. Not only the nominal, expected behavior is learned, but also failing behaviors, i.e., those behaviors leading the system to (several classes of) failure. This "detection-oriented testing" allows improving the overall effectiveness by carrying out an efficient training phase.

3. THE DETECTION FRAMEWORK

3.1 Failure Mode Assumptions

This work focuses on crash failures, i.e., unexpected termination (e.g., due to run-time exceptions), and on hang failures, i.e., the system no longer provides its services or services are delivered unacceptably late. As for hang failures we further distinguish in active and passive hang. The former occurs when a process is still running but other processes may no longer perceive its activity, because one of its threads, if any, consumes CPU cycles improperly. The latter occurs when a process (or one of its threads) is indefinitely blocked, e.g., it waits for shared resources that will never be released (i.e., it encounters a deadlock). Hangs might be either silent or non-silent. In the former case it compromises the communication capabilities of the process, e.g., it cannot reply to heartbeats. In the latter case, the process is still able to communicate, even if the service is not delivered properly. In complex systems it is hard to tell whether a process is currently subject to a passive hang, because it can be deliberately blocked waiting for some work to be performed (e.g., when pools of threads are used in multi-threaded server processes). Difficulties are also encountered with active hangs, because a process (thread) can deliver late heartbeat response, due to stressing workload and working conditions.

3.2 Overall Framework Description

The detection framework is designed to address complex and distributed software systems. The efficacy of the detector depends basically from i) the capacity to uncover errors, ii) from the detection latency and iii) from the false alarm rate. For instance a perfect detector would uncover all the errors with no delays and no false positives.

Assuring software dependability in such kind of systems is typically coped with fault removal techniques in the development phase, most commonly *software testing* techniques, and with fault tolerance mechanisms during operation, which relies on *error*

detection ability and error treatment: typically the planning and implementation of these activities proceed quite independently. Thus, our first aim is to exploit information produced during the testing phase, that in critical systems is intensive, in order to *learn* the system's behavior. In this way, there is no need for a tailored full training phase, because most of the training is carried out during testing. Hence, this guarantees a training that is *i)* more efficient, by reducing the effort for learning, and *ii)* more accurate in the reproduction of the real operational conditions (indeed, this

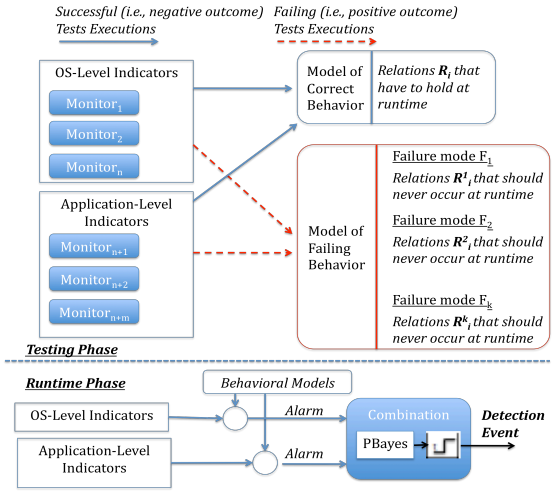

Figure 1. The overall approach.

is a crucial goal during critical systems testing), yielding expectedly better behavior description. The overview of the proposed approach is in Figure 1. A set of monitors is in charge of collecting information during testing, in order to build behavioral models. Thus, during this phase, which can be seen as a training, two kinds of reference models will be constructed: one representing the nominal behavior, and the other representative of the failing behavior.

We represent the behavioral model by means of a set of relations ***Ri*** for each variable ***i*** (denoted as "indicator"), which have to hold at runtime (for correct behavior models) or that should never occur at runtime (for failing behavior models). For instance the relations ***Ri*** for an indicator ***i*** may be a range of values in which the indicator must be included at runtime, or an event (e.g., null return value of a method) that should never occur.

Indicators will be monitored during the execution of all the test cases: if the execution results in the expected outcome, the observed indicator values are used to improve the description of the correct nominal behavior. Whereas, if the test case execution results in a failure, the observed indicator values and the failure mode could be used to populate the failing behavioral model. Failing behaviors are classified by failure modes $\mathbf{F_1 \ldots F_k}$, and a behavioral sub-model (i.e., a set of relations) is associated with each observed failure mode. Typically, the correct behavior has more chances to be learned during the functional system tests, whereas the failing behavior during the robustness/stress test campaigns, which aim making the system fail.

Indicators are monitored again at runtime and if there is an anomaly with respect to the inferred relations $\boldsymbol{R_i}$ of correct behavior, an alarm is triggered by the monitor of the indicator ***i***. Similarly, if monitors observe a relation similar to that inferred for failing behavior model, an alarm is triggered. Alarms generated by monitors of indicators feed the "combination" block, which combines them by means of the *Bayesian Rule* and evaluates a final score indicating the risk of actual failure, as explained in Section 3.5.

3.3 Definition of Indicators and Behavioral Relations

Once the overall approach is defined, the second relevant point is identifying the set of variables we need to describe the system's behavior, i.e., the indicators. We cannot rely on single kind of information, but we need a description based on multiple indicators, which have to depict the system behavior from different perspectives, while keeping at the same time the monitoring overhead low. We identified the following sources of information, some of them taken from the literature [13][14], describing the behavior both at *application level* and at *OS level*:

1) **Interactions pattern** among modules, describing how modules are expected to communicate with each other;
2) **Input/Output invariants** describing the relations holding in the exchanged parameter values among modules (or functions inside the same module);
3) **OS-level information for the expected behavior**, including task scheduling times, waiting and holding times in critical sections, I/O throughput;
4) **Interactions patterns** among modules **and I/O invariants in failing conditions**, to be related with the classes of failures observed during robustness/stress tests;
5) **OS-level information in case of failing behaviors**, such as system calls error code, process and threads exit codes.

Anomalies with respect to these indicators occur when the expected behavior is violated. However, the "expected *behavior*", which is associated with different relations $\boldsymbol{R_i}$'s, has a different meaning depending on the indicator.

For the first indicator, i.e., the interaction patterns, the typical sequence of calls is considered as in [13]; patterns different from the one built during the testing phase cause an alarm triggering. A simple interaction invariant is as follow: for the networking module of the DDS middleware that we experimented (see Section 4.1 for details), zero or more invocations of the discovery service represents an invariant, indicated as *networking_discovery** invariant. For OS-level indicators, a range r_i is considered for each indicator i: if values v_i of the indicator are outside the range, an alarm is triggered. The range is determined by means of static thresholds $[Tl_{vi}, Tu_{vi}]$, such as the minimum and maximum observed values min_{vi} and max_{vi}, computed during test executions. Relations are therefore $Tl_{vi} \leq v_i \leq Tu_{vi}$. For example, in our application scenario (see Section 4), the timeouts/sec expired for the holding time of mutex must be in the range [6,10].

As for indicators related to the I/O invariants, the reference values are built during the invariant construction phase. In this case, the tool adopted to build invariants (described in the next subsection) considers several kinds of relation, not only the one requiring a value to be comprised in a range $[r_{min}, r_{max}]$. For instance, an invariant may involve two variables, and require them to respect the relation $x < y$; similarly, relations observed on three variables, or on sequence of variables may be reported as invariants. Thus, an anomalous behavior is signaled when the current values (of one or more I/O arguments) violate these invariants. For instance, considering the discovery service an I/O human readable invariant is: "discoveryReader->partitions contains no nulls".

Indicators in the points 4) and 5) refer to the same relations of the previous points; but observed in failing conditions. Runtime values should never satisfy those relations.

In order to reduce the occurrence of false-positives in the detection phase we take into account the bursty behavior of some OS monitored indicators and the confidence in the evaluation of I/O invariants. For bursty behavior we mean the sudden occurrence of some events for a short time period (they then disappear). To this aim, we adopt the approach defined in [12] which basically rely on a counting approach: an alarm is triggered when the indicator is anomalous for C consecutive times. Moreover, for I/O invariant indicators, a confidence level associate with invariants can be set in the construction phase (i.e., during tests), which influences the probability of an anomaly value to be a false positive. The lower the confidence parameter, the higher the likelihood that violations of invariants are true positives. In our case, the confidence level is set to *0.01*, meaning that the tool reports invariants that are no more than *1%* likely to have occurred by chance. Note that the relations coming from invariants analysis actually need the availability of the source code, while the OS-derived relations are completely application independent.

3.4 Information Collection

We obtain the OS related indicators by means of the monitoring infrastructure described in [12], which is based on a lightweight interface for kernel modules to implant probes and register corresponding probe handlers. Probes are breakpoints inserted dynamically into the kernel module without modifying the source code. When a breakpoint is hit, a handler routine is launched to provide the needed information (e.g., input parameters or return values of called functions). This does not interfere with program execution, except for a short overhead (see section 4.2).

Two kinds of invariants have been generated, i.e., interaction and I/O invariants. Interaction invariants describe the interaction patterns among components [13], whereas I/O invariants are pre-condition and post-condition, expressed as Boolean expression, on exchanged argument values at the entry/exit point of a function [15]. Invariants are evaluated by means of the analysis of the collected traces exploiting the Daikon tool [16]. In order to check at run-time the invariant, we instrumented the most critical parts of our system, namely the memory and networking management modules, to identify invariants found during testing. When these invariants are violated an alarm is triggered.

3.5 Multi-index Anomaly Detection

As shown in Figure 1, each monitor triggers an alarm in case of anomalies that could potentially result in a system failure. As suggested by intuition, combining several alarms from different sources allows detecting more errors and increasing detection quality, if compared to single monitors. Indeed, monitors are designed to tailor particular class of errors that they will detect better than others. Hence, anomalies resulting from errors that get undetected for a specific monitor, are likely to be detected by another monitor. Outputs of the monitors are therefore fed to a combination block in charge of combining, in a unique "risk value", information coming from single monitors, and issuing a *detection event* (see Figure 1) if such value is greater than a given threshold *Th*. Starting from this detection event and on the alarms received from single monitors the most proper countermeasures can be triggered.

Correlation is based on the Bayes rule that combines information coming from new events with the existing knowledge about the occurrence of a given event. Let us denote with F the "fault activation" event, with $\underline{\mathbf{a}}$ the vector containing the alarms of all the monitors (each value a_i is 1 if an alarm is triggered, 0 otherwise); the probability of the activation of a fault once observed a vector $\underline{\mathbf{a}}$ is:

$$P(F \mid \mathrm{a}) = \frac{P(\mathrm{a} \mid F)P(F)}{P(\mathrm{a} \mid F)P(F) + P(\mathrm{a} \mid \neg F)(1 - P(F))} \qquad (1)$$

Where $P(\mathrm{a}|F)$ represents the probability of detection, being the number of occurrences of $\underline{\mathbf{a}}$ under faulty execution over the total number of vectors collected, whereas $P(\mathrm{a}|\neg F)$ is the probability of false alarms (i.e., the occurrences of $\underline{\mathbf{a}}$ in faulty-free executions). The *a priori* probability of having a system in the "error" state can be either estimated as T/MTTF (i.e., the detection period over the Mean Time To Failure) if previous field data exist (e.g., data of a previous version of the system), or assumed by the literature. In our experiments, we assumed $P(F) = 10^{-6}$ from a literature empirical study [18].

4. EXPERIMENTS

4.1 Case-study and Experimental Setup

In order to investigate the effectiveness of our approach, we have conducted an experimental campaign on a prototype developed at SESM [1], named SWIM-BOX.

Case study description

SWIM-BOX was developed in the framework of the European-wide initiatives aiming at pursuing global interoperability in the Air Traffic Management (ATM) domain, namely the FP6 European project SWIM-SUIT[2]. The proposed case study is actually a pilot prototype for SWIM, the world recognized initiative aiming to enable several ATM stakeholders, i.e., airports, airlines, military air defense, Area Control Centers (ACC) and Air Navigation Service Providers (ANSP), to collaborate by sharing information on a really large scale. The SWIM-BOX is a complex modular Off-The-Shelf-based Java application that relies on facilities provided by the application server JBOSS[3], on data distribution middleware (i.e., *OpenSplice* DDS in the experiments), and on a security manager component (see the prototype architecture in Figure 2).

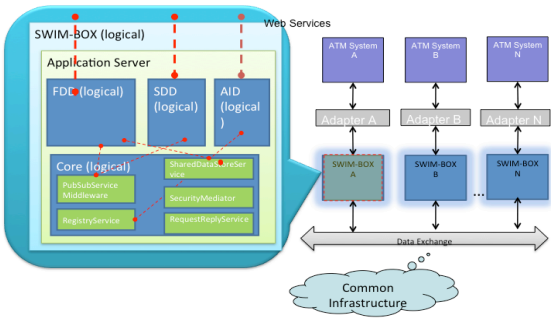

Figure 2. The SWIM-BOX architecture.

[1]SESM s.c.a.r.l. A Finmeccanica company. http://www.sesm.it/
[2]http://www.swim-suit.aero/swimsuit/

Experimental setup

The experimental testbed consists of two legacy entities, named the *Contributor* and the *Manager*, which collaborate to manage *Flight Data Plans*. Both the *Manager* and the *Contributor* run on a cluster node equipped with Intel Xeon *2.5* Ghz (*4* core) CPU, 8GB of RAM, running Red Hat Enterprise Edition 5 Operating System. The detector has been deployed at each node.

Manager and Contributor communicate through a networking infrastructure. However, we focus on a single node of the system to perform detection.

4.2 Experiments Execution and Results Analysis

This section describes the executed experiments and shows the preliminary results. At this stage, the framework implements the learning modules for the Models of Correct Behavior (see Figure 1), by means of indicators both at OS and at application level; currently it still does not implement the learning modules for Model of Failing Behavior. This should be considered when looking at the achieved results because building models of failure behavior could improve detector capability, especially in terms of coverage (see Section 4.2).

We executed two sets of experiments: *golden runs* and *faulty runs*. During golden runs the workload is executed with no artificial fault injected, and they have the aim of characterizing the correct behavior of the system. Faulty runs consist of the injection of a fault and then in the execution of the workload with the goal of generating real failure related data. Thus the former allow building models used for detection, while the latter let evaluate the performance of the detector.

Monitored data were stored in an online data repository so that they can be processed off-line too.

For golden runs, we exploited "ready-to-use" workloads, which were applied to validate the SWIMBOX. Adopted workloads differ from message rate (messages/minutes), message burst rate (bursts/hours) and message per burst. The target system is the same as in the testing phase, except that it is equipped with i) the tracing infrastructure to collect OS-Level indicators and with ii) *daikon* tools to collect Application-Level indicators as described in section 3.4. Thus, these runs emulate the testing phase.

During faulty runs, we apply the same workload of golden runs to the monitored system, but we also inject faults into some components of the data distribution middleware. Referring to the fault classes, defined with respect to the Orthogonal Defect Classification [4], and to the distribution of software faults described in [3], we inject one bug per run using a source code mutation technique. The distribution and the type of the injected faults are provided in Table 1. In fact, faulty runs emulate system failures; they lead to application hangs (60% of total runs) and crashes (20%), and they allow evaluating the performance of the detection framework. It is worth to note that that faulty runs that did not result in a failure despite of the fault activation were not included in the analysis. To estimate how our detection framework deviates from the ideal one described in section 3.2, we evaluate the following well-know metrics [2]:

- **Precision (P)**: the ratio of correctly detected anomalies to the total of all detections (correct and incorrect detections);
- **Recall (R)**: the ratio of correctly detected anomalies to the total of all detectable anomalies (correct detections and not detected);
- **False Alarms rate (FPR)**: the ratio of incorrectly detected anomalies to the total of non-anomalies (correct not detections and incorrect detections).
- **Latency**: the time between fault activation and detection.

Results analysis

Table 1. Classes of injected faults

ODC type	Fault Nature	Fault Type	#
ASSIGNMENT	MISSING	MVIV - Missing Variable Initialization using a Value	3
		MVAV - Missing Variable Assignment using a Value	7
		MVAE - Missing Variable Assignment using a Value	7
	WRONG	WVAV - Wrong Value Assigned to Variable	7
CHECKING	MISSING	MIA - Missing IF construct Around statement	1
	WRONG	WLEC - Wrong logical expression used	2
INTERFACE	MISSING	MPFC – Missing parameter in function call	1
ALGORITHM	MISSING	MFC - Missing Function Call	5
		MIEB - Missing If construct plus statement, plus else before statement	1
TOTAL			34

We first evaluated performance of the detector by exploiting i) only one of the indicators, and then ii) the whole set of indicators evaluated in the testing phase. **Table 2** summarizes the performance of the detector exploiting one indicator per time. Results coming from the combination of all indicators are shown in Table 3. We point out that performance is evaluated

Table 2. Detector performance using just one indicator

Indicators	FPR	P	R
TE/sec for Scheduling Processes	50%	48%	18%
TE/sec for write on disk	25%	23%	9%
TE/sec for read on disk	32%	29%	50%
TE/sec for holding mutexes	10%	91%	100%
TE/sec for waiting mutexes	1%	100%	38%
Process creation per second	5%	94%	100%
TE/sec since last socket read/write	23%	81%	100%
Byte/sec read or write on disk	4%	97%	75%
Byte/sec read or write on network interface	6%	90%	14%
Syscall_err returned per second	8%	60%	20%
I/O-Interaction Invariants	0%	100%	75%
Average	15%	73,9%	54,4%

[3] http://www.jboss.org/

considering different thresholds for the triggering of a detection event. It is clear that a trade-off should be sought in order to have a timely detection. We show, by means of a fault injection campaign, that dependability assessment can leverage data gathered during testing phase successfully by means of a fault injection campaign. Precision and Recall dramatically increase when several indicators are combined, by leaving the detection latency acceptable. We also performed some quantitative analysis by measuring the execution time of our application, in order to evaluate the overhead of the monitored infrastructure. The tracing infrastructure has an overhead of about 2.5% (see Figure 3), in the worst case. The overhead of Daikon tool is very heavy during collection phase (about 25% with our setup) but small, about 5%, at run-time (when checking for invariants violations is performed). Reducing the number of program points or variables can speed up the collection phase for Daikon, even if the accuracy, in terms of the number of invariants found, gets worse.

Table 3. Detector performance using all the indicators

Mean Latency (sec)	Recall	Precision	False Positive rate	Theshold
0,1	100%	98%	2,4%	1,00E-05
1,8375	100%	99%	1,0%	1,00E-04
3,3375	88%	99%	0,8%	5,00E-04

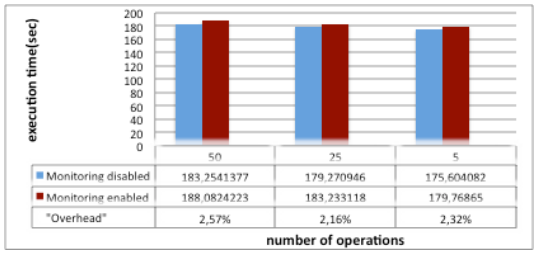

Figure 3. Overhead of the tracing infrastructure

5. CONCLUSIONS AND FUTURE WORK

The detection approach presented in this paper shows that combining several sources of information that can be learned during the testing phase brings to good results in terms of precision, recall, false alarms, and latency. The implementation of failing behavior learning module is the next step to be carried out to further improve these achievements. Moreover, by means of invariants, we plan to investigate also the detection of content failures [1], other than hang and crash. The so-defined approach can be applied either to the overall system, or to one single critical component. In both cases, if proper proactive/reactive actions are defined in case of anomalies (e.g., for anomalies to each learned failure mode), the resulting wrapper would act as a sort of *airbag* for the system/component, since it would prevent it from failing, or from failing unexpectedly. The most challenging, and promising, idea is to let the detection framework learn from the system execution and add a feedback on-line action in charge of re-training the detection infrastructure on the basis of past history in order to face application or environment changes. This would dramatically reduce the false positive rate and increase the detection quality over system lifetime. Finally, we plan to compare the proposed detection framework different frameworks exploiting operational data in order to assess the actual benefit of information collection during testing.

6. ACKNOWLEDGMENTS

This work has ben partially supported by the Italian Ministry for Education, University, and Research (MIUR) in the framework of the Project of National Research Interest (PRIN) "DOTS-LCCI: Dependable Off-The-Shelf based middleware systems for Large-scale Complex Critical Infrastructures".

7. REFERENCES

[1] A. Avizienis, J.C. Laprie, B. Randell, C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. Trans. Dependable Secure Computing, 2004.

[2] F. Salfner, M. Lenk, M. Malek. A survey of online failure prediction methods. ACM Computing Surveys, 2010.

[3] J. A. Duraes and H. Madeira. Emulation of software faults: A field data study and apractical approach. IEEE Trans. on Software Engineering, 32(11):849–867, 2006.

[4] M.Sullivan, R. Chillarege. Software defects and their impact on system availability- A study of field failures in operating systems. 21st Int. Symp. on Fault-Tolerant Computing (FTCS-21), 2–9, 1991.

[5] Wilfredo, Torres. Software Fault Tolerance: A Tutorial. NASA Langley Technical Report Server, 2000.

[6] Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera. On the quality of service of failure detectors. IEEE Trans. Computing, 51(1):13–32, 2002.

[7] A. Thakur, B. K. Iyer. Analyze-NOW-an environment for collection and analysis of failures in a network of workstations. In Proc. of the 7th Intl. Symp. on Software Reliability Engineering, pp 14, 1996.

[8] L. Wang, Z. Kalbarczyk, W. Gu, R.K. Iyer. Reliability microkernel: Providing application-aware reliability in the os. IEEE Transactions on Reliability, 56(4):597–614, 2007.

[9] G. Khanna, P. Varadharajan, S. Bagchi. Automated online monitoring of distributed applications through external monitors. IEEE Trans. Dependable Secure Computing, 3(2):115–129, 2006.

[10] S. Forrest, S. A. Hofmeyr, A. SomayaJi, T. A. Longstaff. A sense of self for unix processes. Security and Privacy, page 120, Washington, DC, USA, 1996. IEEE Computer Society.

[11] S. R. P. Jagadeesh, Chandra Bose, S. H. Srinivasan. Data Mining Approaches to Software Fault Diagnosis. In Proc. of the 15th Workshop on Research Issues in Data Engineering: Stream Data Mining and Applications, pp. 45–52, 2005.

[12] G. Carrozza, M. Cinque, D. Cotroneo, R. Natella. Operating System Support to Detect Application Hangs. In Proc. of the BCS 2nd International Workshop on Verification and Evaluation of Computer and Communication Systems, 2008

[13] L. Mariani, M. Pezzé. Behavior Capture and Test: Automated Analysis of Component Integration. In Proc. of the 10th IEEE International Conference on Engineering of Complex Computer Systems, 2005.

[14] I. Irrera, J. Duraes, M. Vieira, H. Madeira. Towards Identifying the Best Variables for Failure Prediction Using Injection of Realistic Software Faults. In Proc. of the 16th Pacific Rim Int. Symposium on Dependable Computing.

[15] R. Pietrantuono, S. Russo, K. S. Trivedi. Online Monitoring of Software System Reliability. In Proc. of the 2010 European Dependable Computing Conference.

[16] M.D. Ernst, J. Perkins, P.J. Guo, S. McCamant, C. Pacheco, M. Tschantz, C. Xiao. The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program, 2007.