Grid Application Fault Diagnosis Using Wrapper Services and Machine Learning

Jürgen Hofer and Thomas Fahringer

Distributed and Parallel Systems Group
Institute of Computer Science, University of Innsbruck
Technikerstrasse 21a, 6020 Innsbruck, Austria
{juergen,tf}@dps.uibk.ac.at

Abstract. With increasing size and complexity of Grids manual diagnosis of individual application faults becomes impractical and time-consuming. Quick and accurate identification of the root cause of failures is an important prerequisite for building reliable systems. We describe a pragmatic model-based technique for application-specific fault diagnosis based on indicators, symptoms and rules. Customized wrapper services then apply this knowledge to reason about root causes of failures. In addition to user-provided diagnosis models we show that given a set of past classified fault events it is possible to extract new models through learning that are able to diagnose new faults. We investigated and compared algorithms of supervised classification learning and cluster analysis. Our approach was implemented as part of the Otho Toolkit that 'service-enables' legacy applications based on synthesis of wrapper service.

1 Introduction

A portion of todays applications used in High-Performance and Grid environments belongs to the class of batch-oriented programs with command-line interfaces. They typically have long lifecycles that surpass multiple generations of Grid and Service environments. Service Oriented Architectures and Web services became a widely accepted and mature paradigm for designing loosely-coupled large-scale distributed systems and can hide heterogeneity of underlying resources. As re-implementation of application codes is frequently too expensive in time and cost, their (semi-)automatic adaptation and migration to newer environments is of paramount importance. We suggest an approach with tailor-made wrapper services customized to each application. Mapping the functionality of applications to wrapper services requires not only to map input and output arguments, messages and files but also to ensure that the applications behavior is well-reflected. For instance the occurrence of faults may lead to errors that need to be detected, diagnosed propagated via the wrapper services interface, such that clients may react appropriately and handled to prevent larger system failures. In order to recover from failures root causes have to be identified, e.g. unsatisfied dependencies, invalid arguments, configuration problems, expired credentials, quota limits, disk crashes, etc. With increasing complexity of Grids

B. Krämer, K.-J. Lin, and P. Narasimhan (Eds.): ICSOC 2007, LNCS 4749, pp. 233–244, 2007.
© Springer-Verlag Berlin Heidelberg 2007

- growing in size and heterogeneity - this tasks becomes increasingly difficult. Several abstraction layers conveniently shield the user from lower level issues. However these layers also hide important information required for fault diagnosis. Users or support staff are forced to drill down through layers for tracking possible causes. For larger number of failures it then quickly becomes impractical and time-expensive to manually investigate on individual causes by hand.

2 Diagnosing Application Faults

Normally software has been extensively tested before released to production. Nevertheless in large-scale deployments and complex environments such as Grids applications are likely to fail[1]. Common reasons are improper installations or deployments, configuration problems, failures of dependent resources such as hosts, network links, storage devices, limitations or excess on resource usage, performance and concurrency issues, usage errors, etc. Our goal is to provide a mechanism to automatically identify and distinguish such causes. The fault diagnosis process consists of the tasks of error detection, hypothesizing possible faults, identification of actual fault via analysis of application, application artifacts and environment and finally reporting of diagnosis results. Two applications are used throughout this paper: the raytracer POV-Ray [29], an open-source general-purpose visualization application and the GNU Linear Programming Toolkit (GLPK) [28] a software package for solving linear programming and mixed integer programming problems.

2.1 Building Fault Diagnosis Models

Instead of requiring a full formal system specification we provide a set of easy-to-use elements for building fault diagnosis models. They allow developers to describe cases in which their programs may fail and users to describe cases in which their programs have failed in the past. As no knowledge on formal system specification techniques is required we believe our approach is practical and more likely to be applied in the community of Grid users. The diagnosis models are rule-based case descriptions that allow services to perform automated reasoning on the most-likely cause of failures of the wrapped application. Results are then reported to clients. Such diagnosis models are constructed as follows.

1. *Indicators* are externally visible and monitorable effects of the execution of a certain application. We distinguish boolean-valued predicates, e.g. the existence of a certain file or directory, indicators returning strings (StringInd) such as patterns in output, error or log-files, indicators returning reals (RealInd) and indicators performing counting operations (CountInd) such as the number of files in a directory. A few examples are given below

[1] In accordance with Laprie [18] we define a *fault* as the hypothesized or identified cause of an error, e.g. due to a hardware defect; an *error* as a deviation from the correct system state that, if improperly handled or unrecognized, may lead to system *failures* where the delivered service deviates from specified service.

$(\exists file)file$	$extract_stdout(regexp)$	$extract_real_stdout(regexp)$
$(\exists file)dir$	$extract_file(file, regexp)$	$extract_real_stderr(regexp)$
$(\exists regexp)pattern_stdout$	$count_pattern_stdout(regexp)$	$exitCode()$
$(\exists file)((\exists regexp)pattern_file)$	$count_files(regexp)$	$wall_time()$

Next to the set of predefined indicators we allow the use of custom user-provided indicators specific to certain applications, e.g. to verify functional correctness via result checks, error rates, data formats, etc. In some cases runtime argument values are needed as parameters for indicators, e.g. to refer to an output file named via a program argument. Formally we use the $\Theta(argname)$ notation to refer to runtime arguments.

2. A *symptom* is a set of indicators describing an undesirable situation, more concretely the existence of a fault. Symptoms are comparisons of indicators with literal values or comparative combinations of indicators evaluating to boolean values.

$symptom \vdash CountInd|RealInd\{< \mid \leq \mid = \mid \geq \mid >\}\{r|r \in \mathbb{R}\}$
$symptom \vdash CountInd|RealInd\{< \mid \leq \mid = \mid \geq \mid >\}CountInd|RealInd$
$symptom \vdash StringInd\{= \mid \neq\}\{s|s \in string\}$
$symptom \vdash StringInd\{= \mid \neq\}StringInd$
$symptom \vdash Predicate|\neg symptom|symptom \wedge symptom$

Examples for symptoms would be if a coredump file was created, occurrence of the string 'Segmentation fault' in stderr, programs exit code other than zero, output values above some threshold, number of output files below a certain number, etc.

3. *Rules* built on the basis of symptoms allow to reason about fault types. We define rules as implications of the form $(s_1 \wedge s_2 \wedge \ldots \wedge s_n) \Rightarrow u$. Example diagnosis rules for the POV-Ray application are given below.

exit=0 $\wedge \exists file(\Theta(\text{sceneout})) \wedge \neg\exists pattern_stdout(\text{"Failed"}) \Rightarrow$ done_successful
exit=249 $\Rightarrow$ failed_illegal_argument
exit=0 $\wedge \exists file(\Theta(\text{sceneout})) \wedge filesize(\Theta(sceneout)) = 0 \quad \wedge$
$\exists pattern_stdout(\text{"Disk quota exceeded."}) \Rightarrow$ failed_quota
exit=0 $\wedge filesize(\Theta(\text{sceneout})) = 0 \Rightarrow$ failed_disk_quota_exceeded
exit=0 $\wedge \neg\exists file(\Theta(\text{sceneout})) \quad \wedge$
$\exists pattern_stdout(\text{"File Error open"}) \Rightarrow$ failed_file_writing_error
exit=0 $\wedge \exists pattern_stdout(\text{"Got 1 SIGINT"}) \Rightarrow$ failed_received_sigint
exit=137 $\wedge \exists pattern_stdout(\text{"Killed"}) \Rightarrow$ failed_received_sigkill
gramExit=1 $\wedge \exists pattern_gram_log(\text{'proxy is not valid long enough'}) \Rightarrow$ failed_proxy_expires_soon
gramExit=1 $\wedge \exists pattern_gram_log(\text{'couldn't find a valid proxy'}) \wedge$
$\exists pattern_gram_log(\text{'proxy does not exist'}) \Rightarrow$ failed_no_proxy
gramExit=1 $\wedge \exists pattern_gram_log(\text{'proxy does not exist'}) \Rightarrow$ failed_proxy_expired

E.g. the second rule states that the return code 249 unambiguously identifies an illegal argument fault. Failures caused by exceeded disk quota are recognized by an apparently successful return code however in combination with a zero-size outputfile and a certain error message.

4. Finally a set of rules builds a fault diagnosis model. The rules are meant to be evaluated post-mortem, i.e. immediately after the execution terminated, in the specified ordering. If no rule evaluates to true, the fault cannot be identified. Depending on the desired behavior the diagnosis can continue the evaluation if multiple rules are satisfied. The fault is then considered to belong to all found classes.

3 Creating Diagnosis Models Using Machine Learning

With increasing utilization both variety and frequency of faults will increase. Our hypothesis is that given a set of past classified fault events diagnosis models can be learned that are able to correctly classify even unseen novel faults. Fault events are analyzed using he superset of the indicators to build a comprehensive knowledge as trainingset for machine learning. For this purpose an initial set of services is created and deployed. At runtime each fault is analyzed to form a fault event. We investigated on two different learning techniques. In supervised classification learning each fault event in the trainingset has be classified a priori. This is a manual step done by users, service provider or developers. Now the classified training set is used as input to the machine learning procedure that creates new models which are then used to classify faults. The second technique is cluster analysis where the faults do not have to be tagged with class labels but the algorithm partitions the trainingset into groups of fault events that have some degree of similarity.

In order to build the knowledge base each fault incidence has to be analyzed using the superset of indicators. For each detected fault event a tuple of the form $((I \cup T) \times (S \cup F))$ is generated and added to a repository. The tuple contains all relevant information characterizing a certain fault incidence specific to a given application. A set of boolean or numeric indicators $i_i \in I$ such as existence, modification, size, open for reading/writing as detailed above and a set of boolean indicators $t_i \in T$ whether certain regular expression-based patterns (error messages, codes) can be found, are applied to a given set of artifacts created during applications runs. Those artifacts include the standard input/output files associated with each process of and application and the execution environment by $s_i \in S$, i.e. stdout, stderr, system log and log files of resource management system and application-specific input and output files $f_i \in F$. The latter set has to be provided by the user and may be a function of the program arguments.

We selected a set of six well-known supervised classification techniques and three different cluster analysis algorithms [6,21,32] listed in Table 1 and Table 2. The techniques were chosen based on their capabilities to analyze all aspects of

Table 1. Overview on Utilized Classification Techniques

Supervised Classification Learning

OneR (OR)	is an algorithm that produces one-level classification rules based on single attributes. A classification rule consists of an antecedent that applies tests to reason about the consequent.
DecisionStump (DS)	produces simple one-level decision trees. Decision trees follow the divide-and-conquer principle where the problem space is partitioned by outcome of tests until all examples belong to the same class.
Logistic (LG)	is a statistical modeling approach based on logistic regression where coefficient are estimated using the maximum log-likelihood method.
BayesNet (BN)	is a statistical modeling approach producing Bayesian networks in forms of directed acyclic graphs with probabilities over relevant attributes.
DecisionTable (DT)	denotes an algorithm that produces a table consisting of relevant attributes, their values and the prediction class.
J48	is an improved version of the C4.5 decision tree machine learning algorithm. in a decision tree each internal node represents a test on an attribute, branches are the outcomes and leaf nodes indicate the class

Table 2. Overview on Utilized Cluster Analysis Techniques

Cluster Analysis	
k-means (SK)	the number of clusters being sought is defined in the parameter k. then k points are chosen as random cluster centers and instances assigned to closest center. then the new mean is calculated for each cluster. this is repeated until the cluster memberships stop changing.
expectation-minimization (EM)	same basic procedure as k-means algorithm, but calculates the probabilities for each instance to belong to a certain cluster, then calculate the statistical distribution parameters
FarthestFirst (FF)	implements the Farthest First Traversal Algorithm [7] which is a heuristic for approximation of cluster centers designed after procedure of k-means

our trainingsets, their acceptance within the machine learning community and past experience of the authors for similar problems.

4 Implementation

In previous work we discussed the semi-automatic transformation of legacy applications to services for integration into service-oriented environments [8,9,10]. We focused on resource-intensive, non-interactive command-line programs as typically used in HPC and Grid environments and presented the *Otho Toolkit*, a *service-enabler* for *Legacy Applications* $\mathcal{LA}$. Based on formal $\mathcal{LA}$ descriptions it generates tailor-made wrapper services, referred to as *Executor Services* $\mathcal{XS}$. They provide a purely functional interface hiding technical details of the wrapping process on a certain execution platform, the *Backend* $\mathcal{BE}$. Input and output arguments, messages to standard input and output, consumed and produced files are mapped to the $\mathcal{XS}$ interface. Multiple views on the same $\mathcal{LA}$ can be defined to reflect different needs or to ease usage of complex interfaces. The Otho Toolkit generates wrapper service source codes including a build system. Multiple service environments can be targeted and the services may be equipped with application-specific features and generic extensions.

Wrapper services, and especially Executor Services $\mathcal{XS}$ synthesized by the Otho Toolkit, already possess detailed knowledge on the application structure and behavior, control its execution and lifecycle and are aware of input and output arguments, messages and files. Moreover they have the necessary proximity to the execution host for fault investigation. Therefore we chose to address and implement the fault diagnosis as part of the Otho Toolkit and the $\mathcal{XS}$ it creates. All indicators were implemented as generic Bash and Python scripts. We extended Otho Toolkits $\mathcal{LA}$ description to include fault diagnosis models. The Otho Toolkit then generates a custom diagnosis program that evaluates each case using the generic indicator scripts immediately after termination of the application. The diagnosis program evaluates the diagnosis model rule by rule. Indicator results are cached to prevent redundant evaluations. If the $\mathcal{XS}$ uses job submission to a resource management systems the $\mathcal{LA}$ and the fault diagnosis script are submitted as one job to ensure execution on the same resource. In addition to the formal notation introduced before we developed a simple XML-based syntax for representing fault diagnosis models.

```
<fdiag>
  <cause name="successful" status="DONE">
    <exitCode value="0" />
    <fileExists name="|sceneout|" />
    <not><regexpStdout value="Failed" /></not>
  </cause>
  <cause name="illegal argument" status="FAILED">
    <exitCode value="249" />
  </cause>
</fdiag>
```

This shortened example lists two root causes each named and tagged with a post-execution status value. A set of indicators sequentially evaluated with logical conjunction can be given. Elements may be negated by adding a 'not' tag.

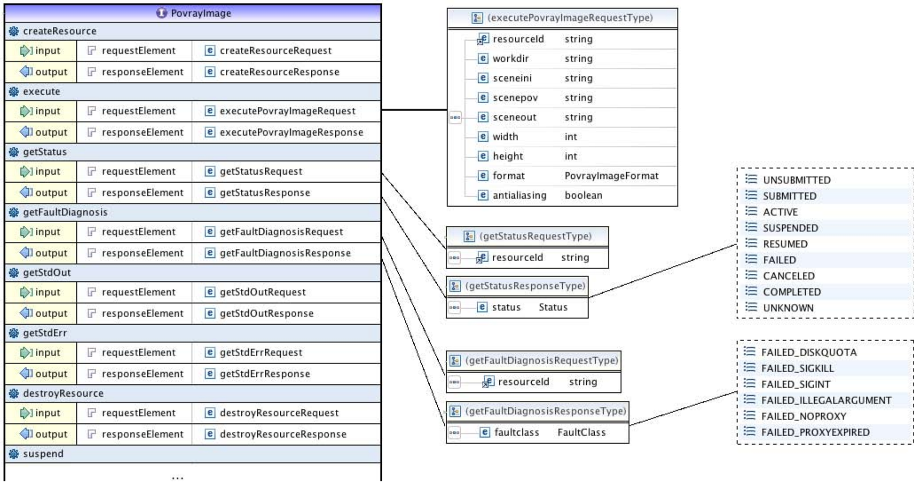

Fig. 1. $\mathcal{XS}$ Interface Adaptations for providing Fault Diagnosis

The fault diagnosis capabilities and states need to be represented in the service interface. Figure 1 shows a partial graphical rendering of the services WSDL interface [31] of synthesized wrapper services for the POV-Ray application and the Axis2 [27] $\mathcal{XS}$ platform. The request type contains the input argument values for the wrapped $\mathcal{LA}$. Operations allow to query service state and fault diagnosis both of which are represented by enumeration values. Obviously the interface differs depending on the service platform used. Axis2 webservice operations for instance carry a job identifier whereas WSRF GT4 services rely on stateful resource properties.

5 Evaluation

For evaluation we used our implementation based on the Otho Toolkit and the $\mathcal{XS}$ it synthesizes. The machine learning techniques described above were implemented as part of $\mathcal{XS}$ using an existing machine learning library [30]. We

deployed both case study applications on the AustrianGrid [26] and injected several types of faults. The resulting training set was used in its raw state ('failed noise'), in a cleaned state ('failed clean') and to allow our classifier to also identify correct behaviour with added successful runs ('failed/succ clean').

The performance or accuracy of classifier is commonly evaluated in terms of their success rates which is the proportion of true and false predictions. An important issue in classification is the question about which set of instances to learn from and which set to evaluate against, as classifiers tend to show better performance if evaluated against the training set than against unseen examples. Therefore we applied three evaluation techniques. First we used the full dataset ('ts') for learning and evaluation. Second we used two-third for learning one-third for evaluation ('66-sp'). Third we used 10-fold cross-validation ('10-cv') where metrics are averaged from ten iterations with 9/10 of examples used for training and 1/10 for evaluation. The set of examples not used for training but to which the classifier is tested against represent unseen fault cases. As we had all fault incidences tagged with class labels the evaluation of the clustering techniques was straightforward. During the learning phase we ignored the classes and then compared the instances in a cluster with their labels counting wrong assignments.

Clustering a trainingset with k attributes spawns a k-dimensional space in which aggregation of examples are to be found. Figure 2 depicts a 2-dimensional subspace with two indicators used in the cluster analysis for faults of the POV-Ray application, namely its exit code and whether a certain pattern occurs in stdout. Elements have been slightly scattered for visualization purposes. The plot nicely illustrates four clusters, three of which are homogeneous. Elements aligned at return code of 137 indicate a 'failed_sigkill' or a 'failed_sighup' signal, depending on the outcome of the second indicator. The 'failed_illegalargument' examples group in this particular view around a return code of 249. Contrarily the other fault incidences cannot be clearly assigned to clusters in this subspace. Figure 3 contains parts of the results of our experiments with the classification learning. Vertical axes show the accuracy. In general it can be observed that prediction accuracy for the GLPK application case study were better than those for the POV-Ray application in most cases. The overall quality apparently strongly

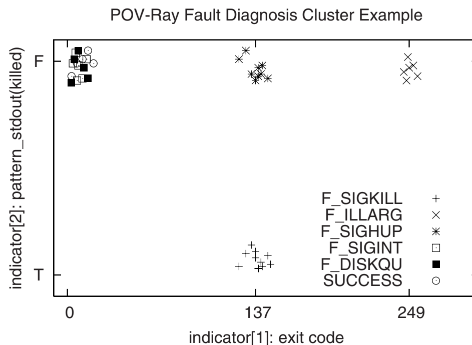

Fig. 2. Visualization of 2-Indicator subspace of SimpleK Means Clustering

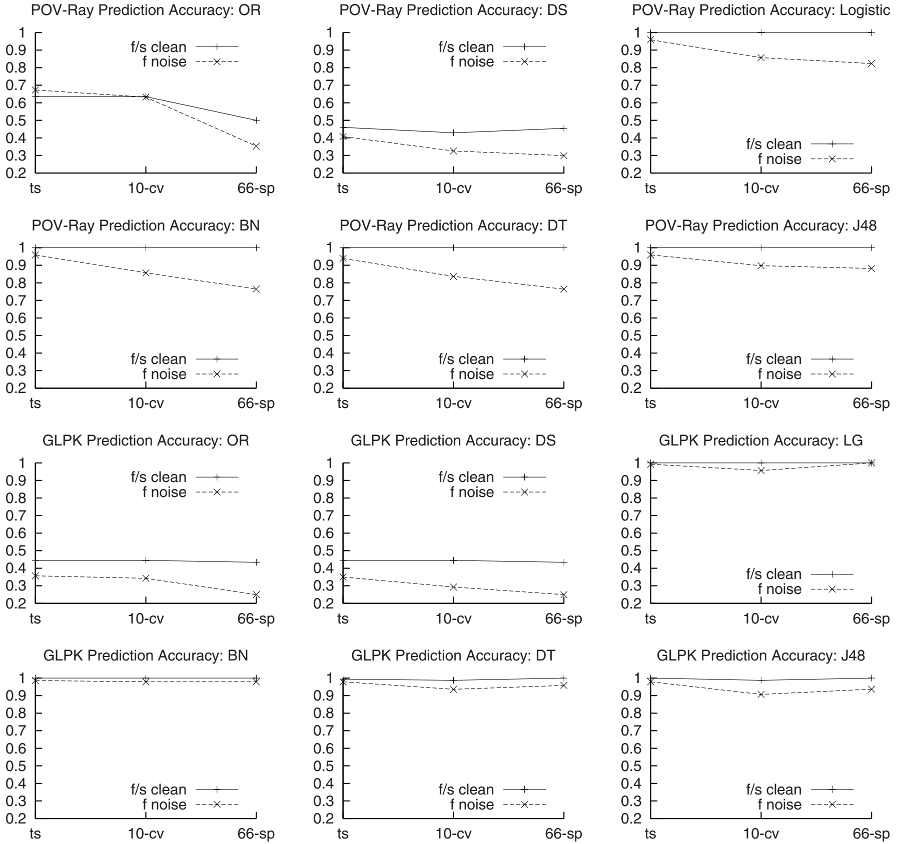

Fig. 3. Evaluation of Machine Learning Algorithms per Algorithm

depends not only on the machine learning technique but also on the concrete application, the indicators used and the corresponding structure of the training set. The second observation is that even on the cleaned datasets the algorithms OR and DS show significantly lower prediction accuracy than LG, DT, BN and J48. For POV-Ray using 10-fold cross-validation DS has an accuracy of only 0.429 and OR of 0.635 on the succ/failed compared to an observed 0.837 lower bound accuracy for the other algorithms. Both methods produce rather simplistic classifier clearly unsuited to capture complex fault events. Nevertheless for trivial diagnostics, e.g. exit code unambiguously identifies fault case, they may be useful as part of meta-models. The remaining four algorithms LG, DT, BN and J48 show comparable performance without significant differences among each other. For the cleaned datasets 'failed/succ cleaned' and 'failed cleaned' all four provide outstanding performance, correctly classifying up to 100% of unseen instances. For instance for the POV-Ray application J48 has on the 'failed uncleaned' raw dataset slightly better performance of 0.898 compared to 0.837 of DT and 0.857 of BN and LG on average on the unseen data. During evaluation we observed

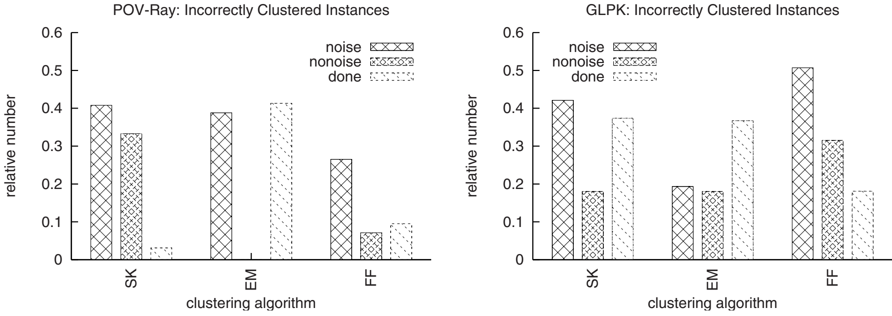

Fig. 4. Evaluation of Clustering Algorithms

that the statistical models BN and LG tend to capture also noise, whereas J48 tree pruning prevented such undesired behavior.

Figure 4 plots some of the results of our evaluation on the clustering algorithms. The plots compares the relative number of incorrectly clustered instances for the algorithms SK, EM and FF for three trainingsets. In general the relative number of errors varied between 0 and 0.5. On the POV-Ray cleaned dataset the EM algorithm was able to perfectly assign all faults to the correct classes. Lower error rates varied up to 0.2. As with our classification algorithms the noisy trainingsets caused major disturbance with error rates between 0.4 and 0.5. In our experiments we could not identify a clustering algorithm significantly outperforming the others. Nevertheless clustering techniques proved to provide good suggestions and valuable guidance for building groups of similar faults. Such otherwise difficult to acquire knowledge and understanding on the nature of faults of applications in a particular setting are of great help to people designing fault-aware application services.

6 Service Evolution for Fault Diagnosis Improvement

The process of lcontinuous diagnosis improvement as realized with the Otho Toolkit and $\mathcal{XS}$ is depicted in Figure 5. Initially a set of services is created and deployed by the Otho Toolkit. At runtime each fault is analyzed, tagged and added to the knowledge base. This is a manual step done by users, service provider or developers. Now the classified training set is used as input to the machine learning procedure that creates new models which enable the classification of unseen fault events that are similar to past faults. The updated or newly learned model is then fed into the Otho Toolkit that creates and redeploys an improved revision of the $\mathcal{XS}$. Additional events are then again collected, learning is re-triggered, followed by synthesis and redeployment and so forth. As the $\mathcal{XS}$ evolves along this cycle its capabilities to diagnose application faults correctly are continuously improved.

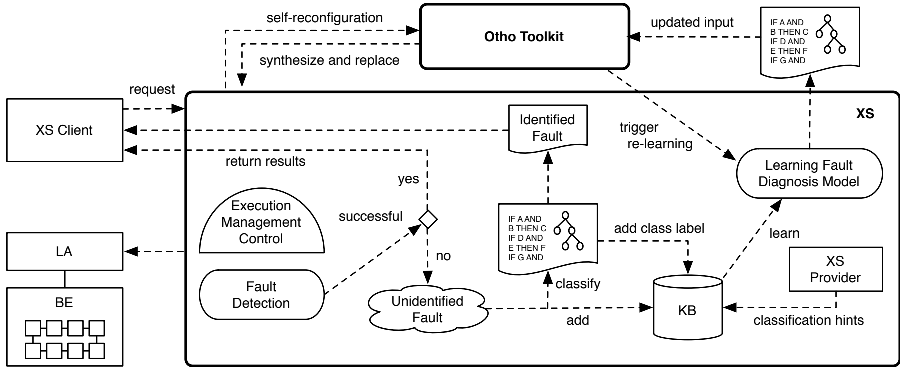

Fig. 5. Service Evolution for Fault Diagnosis Improvement

7 Related Work

Monitoring and failure detection systems [12,14,25] are important Grid components however they discriminate faults no further than into generic task-crashes and per-task exceptions. On the other hand a variety of systems has been suggested for building fault tolerant applications and middleware [13,16] which could benefit from accurate and detailed diagnosis of faults and their causes. Common approaches for fault diagnosis start from formal system specifications [1,15,22] or from its source code [4,17] to derive test cases. Instead neither source code availability nor a formal system specification are prerequisites to our approach. Fault diagnosis in Grids however still is a largely manual time-consuming task. Efforts include an approach for fault localization through unit tests [5] that however requires manual implementation of test cases and frameworks for verification of software stacks and interoperability agreements [24]. Instead we use a model-based description and to automatically generate diagnosis code. The use of machine learning has been successfully applied to many kinds of different classification problems [3,23], e.g. to classify software behavior based on execution data [2] or to locate anomalies in sets of processes via function-level traces [20]. The use of Bayesian Belief Networks for fault localization was proposed [19] but provides neither implementation nor experimental evaluation.

8 Conclusion

With increasing size and complexity of Grids manual application fault diagnosis is a difficult and time-expensive task. We developed a model-based mechanism allowing users, support staff or application developers to formulate precise, rule-based fault diagnosis models evaluated immediately after program termination. Such diagnosis models are used by services to provide accurate and reliable reports. Our approach was implemented as part of application wrapper services synthesized by the Otho Toolkit. In addition we suggest the use of machine learning to semi-automatically create fault diagnosis models based on past classified

fault events. Our evaluation showed that the learned diagnosis models were able to classify novel fault situations with high accuracy. The overall performance of the learned classifier was good but depends on the quality of the dataset. We observed significant perturbation caused by noisy or falsely labeled examples. Ideally developers, service providers and knowledgeable users therefore regularly remove unclean examples from the training set. Our results motivate us to continue with the presented work. We plan to use a larger set of applications to get access to a larger variety of faults. Moreover we intend to investigate on overheads and scalability of our fault diagnosis and machine learning approach.

Acknowledgements

This paper presents follow-up work to a preceding workshop contribution [11]. The presented work was partially funded by the European Union through the IST FP6-004265 CoreGRID, IST FP6-031688 EGEE-2 and IST FP6-034601 Edutain@Grid projects.

References

1. Abrial, J.-R., Schuman, S.A., Meyer, B.: A specification language. In: McNaughten, R., McKeag, R.C. (eds.) On the Construction of Programs, Cambridge University Press, Cambridge (1980)
2. Bowring, J., Rehg, J., Harrold, M.J.: Active learning for automatic classification of software behavior. In: ISSTA 2004. Proc. of the Int. Symp. on Software Testing and Analysis (July 2004)
3. Chen, M., Zheng, A., Lloyd, J., Jordan, M., Brewer, E.: Failure diagnosis using decision trees. In: ICAC. Proc. of Int. Conf. on Autonomic Computing, York, NY (May 2004)
4. Millo, R., Mathur, A.: A grammar based fault classification scheme and its application to the classification of the errors of tex. Technical Report SERC-TR-165-P, Purdue University (1995)
5. Duarte, A.N., Brasileiro, F., Cirne, W., Filho, J.S.A.: Collaborative fault diagnosis in grids through automated tests. In: Proc. of the The IEEE 20th Int. Conf. on Advanced Information Networking and Applications, IEEE Computer Society Press, Los Alamitos (2006)
6. Han, J., Kamber, M.: Data Mining: Concepts and Techniques. Morgan Kaufmann, San Francisco (2001)
7. Hochbaum, Shmoys,: A best possible heuristic for the k-center problem. Mathematics of Operations Research 10(2), 180–184 (1985)
8. Hofer, J., Fahringer, T.: Presenting Scientific Legacy Programs as Grid Services via Program Synthesis. In: Proceedings of 2nd IEEE International Conference on e-Science and Grid Computing, Amsterdam, Netherlands, December 4-6, 2006, IEEE Computer Society Press, Los Alamitos (2006)
9. Hofer, J., Fahringer, T.: Specification-based Synthesis of Tailor-made Grid Service Wrappers for Scientific Legacy Codes. In: Grid'06. Proceedings of 7th IEEE/ACM International Conference on Grid Computing (Grid'06), Short Paper and Poster, Barcelona, Spain, September 28-29, 2006 (2006)

10. Hofer, J., Fahringer, T.: The Otho Toolkit - Synthesizing Tailor-made Scientific Grid Application Wrapper Services. Journal of Multiagent and Grid Systems 3(3) (2007)
11. Hofer, J., Fahringer, T.: Towards automated diagnosis of application faults using wrapper services and machine learning. In: Proceedings of CoreGRID Workshop on Grid Middleware, Dresden, Germany, June 25–26, 2007, pp. 25–26. Springer, Heidelberg (2007)
12. Horita, Y., Taura, K., Chikayama, T.: A scalable and efficient self-organizing failure detector for grid applications. In: Grid'05. 6th IEEE/ACM Int. Workshop on Grid Computing, IEEE Computer Society Press, Los Alamitos (2005)
13. Hwang, S., Kesselman, C.: A flexible framework for fault tolerance in the grid. Journal of Grid Computing 1(3), 251–272 (2003)
14. Hwang, S., Kesselman, C.: Gridworkflow: A flexible failure handling framework for the grid. In: HPDC'03. 12th IEEE Int. Symp. on High Performance Distributed Computing, Seattle, Washington, IEEE Press, Los Alamitos (2003)
15. Jones, C.: Systematic Software Development using VDM. Prentice Hall, Englewood Cliffs (1990)
16. Kola, G., Kosar, T., Livny, M.: Phoenix: Making data-intensive grid applications fault-tolerant. In: Proc. of 5th IEEE/ACM Int. Workshop on Grid Computing, Pittsburgh, Pennsylvania, November 8, 2004, pp. 251–258 (2004)
17. Kuhn, D.R.: Fault classes and error detection in specification based testing. ACM Transactions on Software Engineering Methodology 8(4), 411–424 (1999)
18. Laprie, J.-C.: Dependable computing and fault tolerance: Concepts and terminology. In: Proc. of 15th Int. Symp. on Fault-Tolerant Computing (1985)
19. Meshkat, L., Allcock, W., Deelman, E., Kesselman, C.: Fault location in grids using bayesian belief networks. Technical Report GriPhyN-2002-8, GriPhyN Project (2002)
20. Mirgorodskiy, A.V., Maruyama, N., Miller, B.P.: Problem diagnosis in large-scale computing environments. In: Proc. of ACM/IEEE Supercomputing'06 Conference (2006)
21. Mitchell, T.M.: Machine Learning. McGraw-Hill, Boston (1997)
22. Ortmeier, F., Reif, W.: Failure-sensitive Specification - A formal method for finding failure modes. Technical report, University of Augsburg (January 12, 2004)
23. Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J., Wang, B.: Automated support for classifying software failure reports. In: Proc. of 25th Int. Conf. on Software Engineering, Portland, Oregon, pp. 465–475 (2003)
24. Smallen, S., Olschanowsky, C., Ericson, K., Beckman, P., Schopf, J.M.: The inca test harness and reporting framework. In: Proc. of the ACM/IEEE Supercomputing'04 Conference (November 2004)
25. Stelling, P., Foster, I., Kesselman, C., Lee, C., von Laszewski, G.: A fault detection service for wide area distributed computations. In: Proc. 7th IEEE Symp. on High Performance Distributed Computing, pp. 268–278. IEEE Computer Society Press, Los Alamitos (1998)
26. AustrianGrid, `http://www.austriangrid.at`
27. Apache Axis2, `http://ws.apache.org/axis2/`
28. GNU Linear Programming Kit (GLPK), `http://www.gnu.org/software/glpk/`
29. POV-Ray, `http://www.povray.org`
30. Weka, `http://www.cs.waikato.ac.nz/ml/weka`
31. Web Service Description Language (WSDL), `http://www.w3.org/TR/wsdl`
32. Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations. Morgan Kaufmann, San Francisco (2000)