Proceedings of the

Second International Workshop on Dynamic Analysis (WODA 2004)

Edinburgh, Scotland
25 May 2004

Introduction

Dynamic analysis captures a broad spectrum of program analyses that deal with data produced at program execution time. WODA 2004 brings together researchers and practitioners working in all areas of loosely defined dynamic analysis. The program committee selected 11 high quality papers from 22 submissions. Every paper was reviewed by at least three program committee members.

The workshop papers cover a variety of topics spanning the use and implementation of dynamic analysis techniques: error localization, test case selection and testing effort focus, runtime system monitoring, memory leak detection, temporal logics, performance analysis, and hardware assisted data breakpoints. Dynamic analysis impacts numerous large specialized fields. Real-time system scheduling, consistency analyses of distributed systems, system modeling and garbage collection all have at their core dynamic analyses. Work in dynamic analysis also needs to consider application domains. Different application domains require varying dynamic analyses: embedded systems may not contain enough memory or processing power for online analyses, while critical database systems may not be able to be stopped for offline instrumentation.

All this variety of factors makes dynamic analysis a rich and fascinating research field. The workshop's goal is to bring together researchers in the area to better define the field, share results and ongoing works, and foster collaborations. We think that the most exciting new results occur through cross-fertilization of various research fields. Our goal is to encourage interaction, idea exchange and brainstorming, not to produce a mini-conference. Thus the workshop contains both technical and position paper presentations and discussion periods.

We thank all of the authors who submitted papers for WODA 2004; the program committee and additional reviewers, who provided thorough and thoughtful reviews of the submitted papers; the ICSE workshop organizers; and all workshop attendees. We hope you find WODA 2004 a rewarding and enjoyable experience.

David Evans, University of Virginia

Raimondas Lencevicius, Nokia Research Center, USA

WODA 2004 General and Program Co-Chairs

Workshop website: http://www.cs.virginia.edu/woda2004/

Proceedings of the Second International Workshop on Dynamic Analysis

Table of Contents

Using Static Analysis to Determine Where to Focus Dynamic Testing Effort

Thomas J. Ostrand
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932
ostrand@research.att.com

Elaine J. Weyuker
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932
weyuker@research.att.com

Robert M. Bell
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932
rbell@research.att.com

Abstract

We perform static analysis and develop a negative binomial regression model to predict which files in a large software system are most likely to contain the largest numbers of faults that manifest as failures in the next release, using information from all previous releases. This is then used to guide the dynamic testing process for software systems by suggesting that files identified as being likely to contain the largest numbers of faults be subjected to particular scrutiny during dynamic testing. In previous studies of a large inventory tracking system, we identified characteristics of the files containing the largest numbers of faults and those with the highest fault densities. In those studies, we observed that faults were highly concentrated in a relatively small percentage of the files, and that for every release, new files and old files that had been changed during the previous release generally had substantially higher average fault densities than old files that had not been changed. Other characteristics were observed to play a less central role. We now investigate additional potentially-important characteristics and use them, along with the previously-identified characteristics as the basis for the regression model of the current study. We found that the top 20% of files predicted by the statistical model contain between 71% and 85% of the observed faults found during dynamic testing of the twelve releases of the system that were available.

Keywords: Software Faults, Fault-prone, Prediction, Regression Model, Empirical Study, Software Testing.

1. Introduction and Earlier Work

Much of today's industry relies on software systems, and requires that they behave correctly, perform efficiently, and can be produced economically. For these reasons, it is important that we dynamically test systems to identify faults residing in the code. For large systems, this can be a very expensive and difficult process. Therefore, we want to determine which files in the system are most likely to contain the largest numbers of faults that lead to failures and prioritize our testing effort accordingly. In that way we minimize the cost of testing and maximize the effectiveness of the process. In order to do this, we have been investigating how to use data residing in a combined version control and change management system used during all stages of development, testing, and field release, to improve dynamic testing.

Preliminary work was reported in an earlier paper [10] which described a case study involving an industrial inventory tracking system, developed over a three year period, covering twelve quarterly releases. The goal of that research was to do static analysis to identify structural characteristics that are associated with files that contain particularly large numbers of faults as determined by reported failures. The data used for the static analysis resides in the combined version control/change management system with some of the data determined by statically analyzing the code while other data were identified during the dynamic testing phase.

Data in this repository were collected during each of nine development phases including requirements, design, development, unit testing, integration testing, system testing, beta release, controlled release, and general release. In this paper we will describe the use of this information to develop a statistical model to predict where faults are most likely to reside in the code, which in turn can be used as an integral part of the dynamic testing process. Thus our process relies on a complex interplay between static and dynamic analysis, and data associated with both of these types of analysis.

Our earlier studies considered the extent to which faults clustered in a small proportion of files, and looked at file characteristics such as size, age, whether the file is new to the current release, and if not, whether it was changed during the prior release, the number and magnitude of changes made to a file, the number of observed faults during early releases, and the number of faults observed during early development stages.

Most of the previous research in this area, including our earlier work [10], and that by other authors described in [1, 2, 3, 5, 8, 9], was aimed at examining software systems to establish *characteristics* that may be associated with high incidences of faults. In this paper, we go beyond merely identifying characteristics and successfully build a statistical model that can *predict* the incidence of faults in future versions of a system. Specifically, this model is used to predict the number of faults that will occur in each file during the next release, based on current characteristics of the file and its behavior in earlier releases. By selecting the set of files that are predicted to account for a large percentage of the faults in the next release, we can encourage testers to use that information to prioritize and focus their (dynamic) testing efforts.

Thus our goal is to accurately identify a relatively small percentage of the files that contain a large percentage of the faults. Of course, there is no guarantee that all faults, or even the most dangerous faults, will be located by this approach. However, if the prediction allows a large majority of all outstanding faults to be identified more rapidly than they would otherwise be found, then more resources will be available for additional testing to ferret out the remaining ones, or the process can be completed more quickly, and hence cheaply, with equal success.

The work by Graves et al. [4] is most similar to ours, as they also construct models to predict fault-proneness. In contrast to Graves et al., however, our model makes predictions for individual files of the system, rather than for modules that are collections of files as was done in [4]. The fact that the granularity of the entities we use in our static analysis is significantly finer than that used by Graves et al. is important since it should facilitate the identification of faults in a much more localized portion of the code, thereby making debugging easier as well.

Other differences between our work and that done by the Graves et al. group include the fact that they attempted only a single prediction while our case study makes predictions for each release beginning with Release 3, and continuing through Release 12, allowing us to validate the effectiveness of our model over a sustained period of time, with the system at different levels of maturity. Also, their models use the fault history of a single two-year period to predict faults in the following two-year period, while our model uses data from much shorter 3-month intervals to predict faults in the following quarterly releases. This shorter interval provides much more timely information to testers, who can use the prediction from the current and prior releases to help focus their testing efforts. In fact the goal of our work is to design a process that can be used as a standard part of the development process in an industrial environment to improve and streamline the testing of systems requiring very high reliability.

Our earlier study considered a file's *fault density*, computed in terms of faults per thousand lines of code (KLOCs). In Section 3 of this paper we will describe our findings related to several new questions regarding the number of faults in a file. Among the new factors we consider is whether there was a relationship between the complexity of the file and the number of faults in a file, where complexity is measured by the cyclomatic number [6] rather than the number of lines of code. We also investigate the role of the choice of programming language, the fault history in the file during the previous release, and the amount of change during the previous release.

As mentioned above, our ultimate goal is to be able to identify a particular set of files in a new release that are determined by our statistical model to be the most likely ones to account for the largest numbers of faults. Since we have determined in our earlier study that faults typically have a highly skewed distribution, this should be possible to accomplish.

The remainder of the paper is organized as follows: In Section 2, we describe the software system that is the subject of our case study and present some basic information about file characteristics and the faults identified during testing. Section 3 illustrates associations between selected file characteristics and the number of faults identified during a particular release. In Section 4 we present findings from a negative binomial regression model to predict the number of faults, in order to analyze relationships while controlling for other characteristics. Finally, Section 5 presents conclusions and describes plans for extending this work.

2. The System Under Study

The system used in this study is the same inventory tracking system as was used during the preliminary study [10]. As a standard part of the operating procedure for most development projects at AT&T, whenever any change is to be made to a software system, a Modification Request (MR) is entered in the combined version control and change management system. Each MR includes information describing the file(s) to be changed, the nature of the change (for example, is this a new file being added, or a modification of an existing one), the details of the change including specific lines of code to be added, deleted, or changed, a description of the change, and a severity indicating the importance of the proposed change. These data are collected as part of the normal development process and were therefore available for every release of the system. It is these data that we will statically analyze in order to use it to streamline dynamic testing.

Some parts of the MR, such as the severity rating, are highly subjective, and therefore may not be particularly useful. Unfortunately, the standard MR format does not require the person initiating the request to indicate whether

Rel	Number of Files	Lines of Code	Mean LOC	Faults Detected	Fault Density	System Test and Later Fault Density
1	584	145,967	250	990	6.78	1.49
2	567	154,381	272	201	1.30	0.16
3	706	190,596	270	487	2.56	0.45
4	743	203,233	274	328	1.61	0.17
5	804	231,968	289	340	1.47	0.19
6	867	253,870	293	339	1.34	0.18
7	993	291,719	294	207	0.71	0.10
8	1197	338,774	283	490	1.45	0.25
9	1321	377,198	286	436	1.16	0.16
10	1372	396,209	289	246	0.62	0.09
11	1607	426,878	266	281	0.66	0.21
12	1740	476,215	274	273	0.57	0.15

Table 1. System Information

the change is due to a fault correction or to some other reason such as performance improvement, cleaning up the code, or changed functionality. We have now succeeded in getting the MR form changed to include a field that explicitly states whether the MR was due to the identification of a fault, but this was not available at the time that the data described here were entered or collected, and so we needed a way of making that determination.

Since our study included a total of roughly 5,000 faults, and many more MRs that were categorized as being other sorts of changes, it was impossible to read through every MR to make that determination. We therefore needed a heuristic and used a rule of thumb suggested by the testing group that an MR likely represents a fault correction if either exactly one or two files were modified. In an informal attempt to validate this hypothesis, we sampled a small number of MRs by carefully reading the text description of the change. In the small sample space, nearly every MR that modified one or two files was indeed a fault fix, and every MR that modified a larger number of files (sometimes as many as 60 files) was not a fault correction, but rather a modification made for some other reason. For example, if a new parameter was added to a file, every file that called it had to be modified accordingly.

Changes can be initiated during any stage from requirements through general release. For most development environments, change recording begins with integration or system test, when control leaves the development team and moves to an independent testing organization. For this system, however, MRs were written consistently from requirements on. Almost three quarters of the faults included in this study were identified during unit testing done by developers.

The final version of the system used in these studies (Release 12) included more than 1,700 separate files, with a total of more than 476,000 lines of code. Roughly 70% of these files were written in java, but there were also small numbers of shell scripts, makefiles, xml, html, perl, c, sql, awk, and other specialized languages. Non-executable files such as MS Word, gif, jpg, and readme files were not included in the study.

Over the three year period that we tracked this system, there was a roughly three-fold increase in both the number of files and lines of code. At the same time, there was a significant concentration of identified faults in files, going from appearing in 40% of the files in Release 1 to only 7% of the files by Release 12. One might hypothesize that the increased fault concentration was simply a reflection of the fact that the system was three times larger. However, when the absolute numbers of files containing faults was considered, this fault concentration was also apparent. For example, in Release 1, a total of 233 (of 584) files contained any identified faults, by Release 8 only 148 (of 1197) files contained any identified faults, and by Release 12, only 120 (of 1740) files contained any identified faults at all.

One important decision that had to be made involved exactly how to count the number of faults in a file. If n files were modified as the result of a failure, then this was counted as being n distinct faults. This is consistent with the convention used in References [8] and [3]. This implies that each fault was associated with exactly one file.

Table 1 provides summary information about the first twelve releases of the system, including lines of code and faults. New files typically represent new functionality, while changed files generally represent fault fixes. As the system matured and grew in size, the number of faults tended to fall, with the largest decrease occurring from Release 1 to Release 2. As one might expect, there is also a general downward trend in the fault density as the system matured, with some exceptions including Release 2.

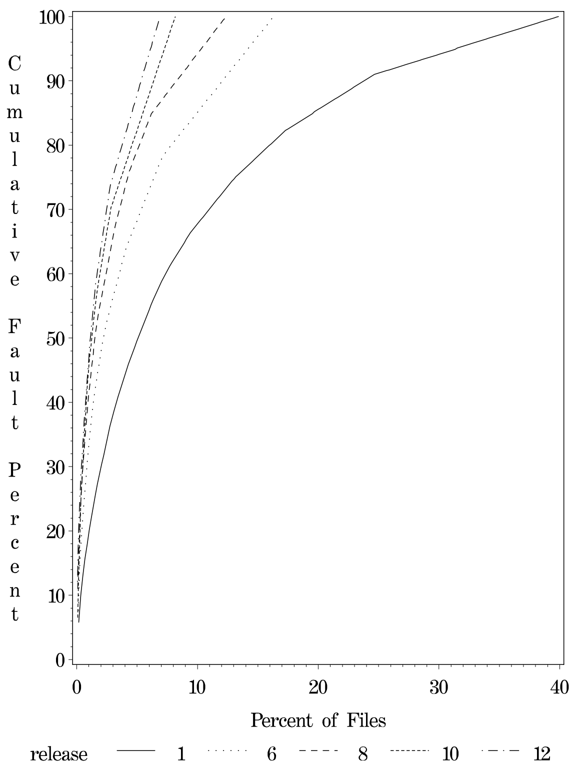


Figure 1. Fault Distribution for Releases 1, 6, 8, 10, 12

The large dip at Release 2 likely occurred because it was an interim release. While other releases all occurred at roughly three month intervals, Release 2 occurred between Releases 1 and 3, which were themselves done three months apart. This likely led to a decreased number of faults identified during Release 2, and hence a decreased fault density.

The last column of the table restricts attention to those faults identified during system test or later. As mentioned above, it is uncommon for faults identified during earlier stages of development to be included in a fault-reporting system. Therefore, the system test fault densities are likely to be more comparable to fault densities reported in other empirical studies. Recall too, that for this system, generally one quarter or fewer of the faults at any release were identified during system test or later.

3. Fault Concentration and Potential Explanatory Factors

In this section we discuss various potential additional factors not considered in our earlier work that might explain the differential fault concentration in files. Once these factors are understood, we will use them to build a statistical model that statically analyzes the software, to guide its dynamic testing.

3.1 Concentration of Faults

Ostrand and Weyuker [10] reported that faults for this system tended to concentrate in a relatively small proportion of files at every release. We repeat here Figure 1 which originally appeared in [10], showing the concentration of faults in Releases 1, 6, 8, 10, and 12. For clarity, we showed data for only a sampling of the releases. The selected releases are representative of the other releases that were not included. We found that when too many releases were shown on the same graph, it became impossible to distinguish among the lines and therefore the import of the data was lost.

The files in each release are sorted in decreasing order of the number of faults they contain. A point (x, y) on the Release R curve represents the fact that x% of the files in Release R contain y% of the faults. For example, at Release 1, the ten percent of files with the most faults (58 files) had 669 faults, representing 68% of the total for Release 1. The curves show that the proportion of faults tends to become increasingly concentrated in fewer files as the system matures.

3.2 File Size

In [10], we examined the relationship between file size and fault density and found that there was a tendency for

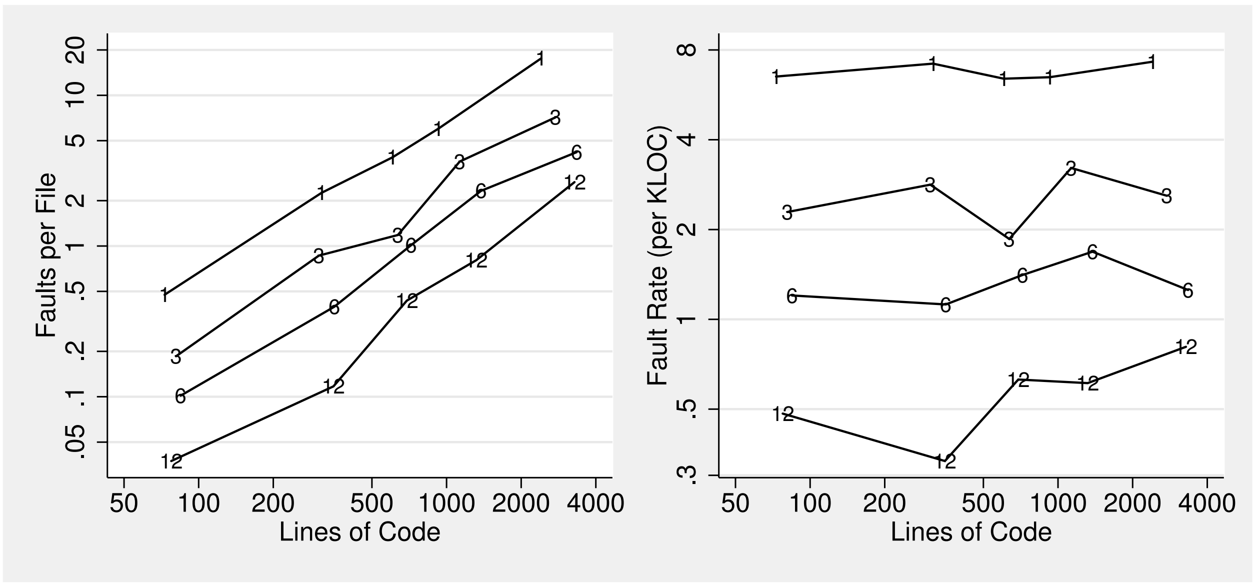


Figure 2. Faults and Files Grouped by Size

small files to have higher average fault densities than large files. We now consider the relationship between file size and the average number of faults per file, rather than considering the average fault density. This will be done by dividing the files for each release into bins. For a given release, we sort the files in increasing size order, divide all of these files into five bins with roughly equal numbers of total lines of code, and calculate the average faults per file for files in the bin.

For example, at Release 1, the first bin contains 398 files ranging in size from 5 to 202 lines of code, with an average size of 73 lines. Those files have a total of 189 faults, resulting in an average of 0.47 faults per file. Subsequent bins include progressively smaller numbers (94, 48, 32, and 12) of larger files, with increasingly more faults per file. This relationship is shown on a log-log scale for Releases 1, 3, 6, and 12. As noted earler, these releases were representative of all releases and were selected to show releases at various stages of maturity. The lefthand portion of Figure 2 shows that there is a strong relationship between file size and the average number of faults per file. We also look at the fault density to see whether there are a "disproportionate" number of faults that occur in larger files than smaller ones, and if there are, whether it might make sense to limit the permitted size of files.

The righthand portion of Figure 2 shows fault densities versus file size for the same sets of bins and releases. The figure shows that there is little or no relationship between fault density and file size. Graphs for the releases not shown in this figure tell similar stories. Although the fault densities for a given release tend to be higher for the two bins containing the largest files than for the two bins containing the smallest ones, the relationship is not monotonic for any of the twelve releases. Specifically, across the releases, the bin containing the largest files has the highest fault density for only five of the twelve releases, and the bin containing the smallest files has the lowest fault density for only three of the twelve releases. Moreover, when results are aggregated across releases, fault densities for largest files are only about 20% higher than for the shortest files. We therefore conclude that file size is not a strong predictor of fault density, but might be a good predictor of the absolute number of faults in a file.

Note that there are two points of difference from our earlier analysis of fault density and size. First, in the present study, we look at the fault density data aggregated over files in a given size range rather than considering each file individually. Second, we include all files: those for which faults were detected, and those for which there were no faults detected. In the earlier work the fault density was computed only for those files that contained faults.

3.3 Program Type

Table 2 compares fault densities for the most commonly-used program types in this system. Because fault densities are much lower for existing files, this table only includes results for a file at the time of its first entry into the system (new files). The observed fault densities vary by a factor of close to 30, with makefiles having the highest average density and xml files the lowest.

Type	Files	LOC	Faults	Fault Density
makefile	94	2509	58	23.12
sh	140	7756	69	8.90
sql	80	6875	60	8.73
html	52	5639	22	3.90
java	1492	413420	1424	3.44
perl	68	17619	52	2.95
c	21	5824	8	1.37
xml	95	5070	4	0.79

Table 2. Fault Densities for New Files, by Program Type

4. Multivariate Analysis of the Number of Faults

In this section we present results from negative binomial regression models that predict the number of faults in a file during a release, as a function of various file characteristics. This modeling process serves three major purposes. First, it provides information about the association between the number of faults and individual file characteristics while holding other file characteristics constant. Most of this information is determined by statically analyzing the code. Information about fault counts is, of course, determined by dynamic execution of the code, primarily on test cases, but also during field operation. Data provided in [10] showed that most faults were detected during either unit or system testing, with only 2% of the faults detected during field operation. The second purpose of the modeling process is to provide a measure of the concentration of faults beyond what is accounted for by file characteristics. This allows us to compare the effectiveness of alternative sets of factors. Third, the model produces predictions of the most fault-prone files in a release, so that testing resources can potentially be targeted more effectively. The third purpose is the ultimate goal of this research. In Section 4.1, we outline the model, while in Section 4.2, we describe our findings. In Section 4.3, we assess the efficacy of this strategy.

4.1 The Negative Binomial Regression Model

Negative binomial regression is an extension of linear regression designed to handle outcomes like the number of faults [7]. It explicitly models counts or outcomes that are nonnegative integers. The expected number of faults is assumed to vary in a multiplicative way as a function of file characteristics, rather than in an additive relationship. Unlike the related modeling approach, Poisson regression, the negative binomial model allows for the type of concentration of faults apparent in Figure 1, in which we see a relatively small percentage of files containing a large percentage of faults. This is done by adjusting inference for the additional uncertainty in the estimated regression coefficients caused by overdispersion.

Let y_i equal the observed number of faults and x_i be a vector of characteristics for file i. The negative binomial regression model specifies that y_i, given x_i, has a Poisson distribution with mean λ_i. This conditional mean is given by $\lambda_i = \gamma_i e^{\beta' x_i}$, where γ_i is itself a random variable drawn from a gamma distribution with mean 1 and unknown variance $\sigma^2 \geq 0$. The variance σ^2 is known as the dispersion parameter, and it allows for the type of concentration we observed for faults. The larger the dispersion parameter, the greater the unexplained concentration of faults. However, to the extent that this concentration is explained by file characteristics x_i that are included in the model, the dispersion parameter will decline.

4.2 Results

We used a negative binomial regression model fit to files from Releases 1 to 12 with the unit of analysis being a file-release combination. This yielded a total of 12,501 observations. The outcome is the number of faults predicted to be associated with the file at the given release. All models were fit by maximum likelihood using the procedure Genmod in SAS/STAT Release 8.01 [11].

Predictor variables for the model are: the logarithm of lines of code; whether the file is new, changed or unchanged (the file's change status); age (number of previous releases the file was in); the square root of the number of faults in the previous release (if any); program type; and release. Logged lines of code (LOC), file age, and the square root of prior faults are treated as continuous variables. File change status, program type, and release are treated as categorical variables, each fit by a series of dummy (0-1) variables, with one omitted category that serves as the reference. For file change status, the reference category is unchanged files, so that the new and changed coefficients represent contrasts with existing, unchanged files. For program type, the reference category is java files, the most commonly-occurring

Predictor Variables	Dispersion Parameter	Amount Explained	Percentage Explained
Null	13.38	NA	NA
LOC	5.61	7.77	58.0
Release	11.00	2.38	17.8
File Change Status	7.29	6.09	45.5
Program Type	12.88	.51	3.8
Prior Faults	9.86	3.53	26.3
LOC, Release	3.91	9.47	70.8
LOC, Release, File Change Status	3.03	10.35	77.4
LOC, Release, File Change Status, Program Type	2.52	10.87	81.2
Full Model	2.27	11.11	83.0

Table 3. Estimated Dispersion Parameters Associated with Selected Models

Release	3	4	5	6	7	8	9	10	11	12
% Faults Identified	77	74	71	85	77	81	85	78	84	84

Table 4. Percentage of Faults Included in the 20% of the Files Selected by the Model

type for this system. We arbitrarily set Release 12 as the reference release.

The strongest predictor in the model is the number of lines of code. Because the model uses the logarithm of lines of code, a coefficient of 1.00 would imply that the expected number of faults grows proportionally with lines of code (i.e., that fault density is unrelated to lines of code). The estimated coefficient was 1.047 which exceeds 1.00. This therefore provides some evidence that fault density grows with lines of code, holding all else equal. We note, however, that the 95 percent confidence interval does include 1.00.

For categorical predictors, each coefficient estimates the difference in the logarithm of the expected number of faults for the corresponding category versus the reference category. For example, for changed files, the coefficient was 1.066. This indicates that changed files have about exp(1.066) = 2.90 times more faults than existing, unchanged files with otherwise similar characteristics. Of course, the changed files are more likely to have other characteristics (such as prior faults) indicating a propensity for faults at the current release.

Table 3 displays estimates of the dispersion parameter for a series of alternative models, to help show the relative improvement associated with individual, or groups, of predictor variables. The estimated dispersion parameter for a null model, with no predictors, is 13.38. The best single predictors were lines of code and the file's change status. Lines of code reduced the dispersion to 5.61, a reduction of 58.0%, while file change status explained 45.5% of the dispersion. Use of the full model reduced the dispersion parameter to 2.27, a reduction of 83.0%.

Various other potential predictor variables were tested, but dropped from the model because they did little to improve the predictive power when added to the model. Some of the variables that we decided to exclude because they did not significantly improve the predictive capability of the model included: the number of changes for files that changed since the previous release, whether or not the files had changed prior to the previous release, and the logarithm of the cyclomatic number (which was computed for java files only). The *cyclomatic number* measures the complexity of a file by counting the number of decision statements in the file [6]. It has been found to be very highly correlated with the number of lines of code. Although the cyclomatic number did predict faults well in a bivariate context, it helped very little when used in conjunction with lines of code (both logged), especially at later releases. In contrast, lines of code remained important even in conjunction with the cyclomatic number.

4.3 Targeting Fault-Prone Files for Testing

We now evaluate the potential of the regression model to improve testing productivity by prospectively identifying a subset of files that contain disproportionately many of the faults at the next release. At each release, beginning with Release 3, we created predictions based on fitting alternative models using data from only the previous releases (e.g., predictions for Release 3 used data from Releases 1 and 2). For each release, these predictions are used to order the files from most to least fault-prone, based on the predicted numbers of faults.

Table 4 shows the percentages of actual faults contained in the top 20 percent of files identified by the full model at each of Releases 3 to 12. The model prospectively identified between 71% and 85% of the faults in the system, with an average over all releases of 80%. Of course any percentage of the files could have been selected, but we determined that 20% was a good choice providing a large percentage of the faults while focusing on a relatively small percentage of the files.

5. Conclusions and Future Work

We have used static analysis to develop a negative binomial regression model as a way of predicting which files are most likely to contain the largest numbers of faults in a new release, and thereby prioritize effort during dynamic testing. This prediction was done for each release by using only data collected during earlier releases. Our initial model was quite successful in the sense that we were able to use it to accurately predict, on average, the 20% of the files that corresponded to 80% of the faults.

The factors that influenced our predictions include the size of the file, the file's change status, the number of faults in the previous release, the programming language, and the file's age. Unlike Graves et al. [4], we found that change history before the prior release was not needed in our models. This finding may be because our models are more specific in terms of content and time since we predict faults for individual files during a series of releases. Graves et al., in contrast, modeled faults for modules which are large groups of files, during a single two year period.

So far we have designed our model based on the characteristics identified as most relevant for the twelve releases of one software system. Although this is a substantial system that runs continuously, with quarterly new releases, there may be characteristics of this system that are atypical, and therefore the model may not be applicable to other systems without tuning. In addition, as the system ages, the most important factors may change somewhat. For this reason, it is important to apply our model to additional releases of the inventory tracking system, as well as to other systems with different characteristics, developed in different environments.

We have now collected data for an additional five releases of the current system, and identified two additional industrial software systems, each with multiple releases and years of field exposure, for which data collection and analysis have begun. Once this is complete, we will apply the current negative binomial regression model to the data collected from these systems and see whether the prediction is as successful as we observed for the first twelve releases of this system. If not, we may have to identify additional relevant characteristics or modify the role played by the factors by defining new weightings. We are also designing a tool to automate the application of our prediction model.

We consider our initial results extremely promising and look forward to the routine use of this sort of predictive modeling to focus software testing efforts, thereby improving both the efficiency and the effectiveness of our software testing process. We have found that using static analysis to guide and prioritize dynamic software testing is an excellent way of improving the testing process for this system.

References

[1] E.N. Adams. Optimizing Preventive Service of Software Products. *IBM J. Res. Develop.*, Vol28, No1, Jan 1984, pp.2-14.

[2] V.R. Basili and B.T. Perricone. Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM*, Vol27, No1, Jan 1984, pp.42-52.

[3] N.E. Fenton and N. Ohlsson. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Trans. on Software Engineering*, Vol26, No8, Aug 2000, pp.797-814.

[4] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy. Predicting Fault Incidence Using Software Change History. *IEEE Trans. on Software Engineering*, Vol 26, No. 7, July 2000, pp. 653-661.

[5] L. Hatton. Reexamining the Fault Density - Component Size Connection. *IEEE Software*, March/April 1997, pp.89-97.

[6] T.J. McCabe. A Complexity Measure. *IEEE Trans. on Software Engineering*, Vol2, 1976, pp.308-320.

[7] P. McCullagh and J.A. Nelder. Generalized Linear Models, 2nd Edition, Chapman and Hall, London, 1989.

[8] K-H. Moller and D.J. Paulish. An Empirical Investigation of Software Fault Distribution. *Proc. IEEE First Internation Software Metrics Symposium*, Baltimore, Md., May 21-22, 1993, pp.82-90.

[9] J.C. Munson and T.M. Khoshgoftaar. The Detection of Fault-Prone Programs. *IEEE Trans. on Software Engineering*, Vol18, No5, May 1992, pp.423-433.

[10] T. Ostrand and E.J. Weyuker. The Distribution of Faults in a Large Industrial Software System. *Proc. ACM/International Symposium on Software Testing and Analysis (ISSTA2002)*, Rome, Italy, July 2002, pp.55-64.

[11] SAS Institute Inc. SAS/STAT User's Guide, Version 8, SAS Institute, Cary, NC, 1999.

Deriving State-Based Test Oracles for Conformance Testing

James H. Andrews
Department of Computer Science
University of Western Ontario
London, Ontario, CANADA N6A 5B7

Abstract

We address the problem of how to instrument code to log events for conformance testing purposes, and how to write test oracles that process log files. We specifically consider oracles written in languages based on the state-machine formalism. We describe two processes for systematically deriving logging code and oracles from requirements. The first is a process that we have used and taught, and the second is a more detailed process that we propose to increase the flexibility and traceability of the first process.

1. Introduction

Testing can be made more automated and reliable by the use of *test oracles*, programs that check the output of other programs. In situations where it is infeasible to capture program input/output directly, the software under test (SUT) must write text log files of events. The oracles that process these log files are then referred to as log file analyzers.

In contrast to other dynamic analysis tasks, the SUT instrumentation needed for writing log files is often dependent on the requirements, and thus cannot be added automatically. The same is true for the log file analyzers. We are therefore faced with the problem of how to insert logging instrumentation manually and how to write log file analyzers. This paper addresses these problems.

We have recently been studying a method of log file analysis (LFA) in which oracles are written in a language (LFAL) based on state machines. Our experiences of writing these oracles have led to a recommended process for moving from requirements to oracles and logging instrumentation. We describe this process in Section 3 below, and report on our experiences of using and teaching it. We have noticed some deficiencies in this process, however, and for this workshop we propose a new, more detailed process (Section 4) with the advantages of greater flexibility and traceability.

We do not believe that LFA testing can or should replace all traditional verification and validation activities, but rather that it can act as a complement to traditional methods, enhancing them by enhancing the reliability of test result checking. We take account of this in our proposed processes. Section 5 discusses the potential benefits and problems of LFA testing, especially with reference to the proposed development processes. We begin, however, with a discussion of the background of this paper.

2. Background

Testing involves selecting test cases, running the test cases on the SUT, and checking the results. This paper deals primarily with test result checking. Here we answer the questions of why test oracles are necessary, what advantages we get by running test oracles on log files, and what motivated our decision to create a state-based language for writing log file analyzer test oracles. We also describe the log file analysis language LFAL and discuss work related to this paper.

2.1. Why Test Oracles?

Test oracles [18] are needed in several common situations. The first is when test output is too complex to be checked by a human. This is the case for applications as diverse as communication protocol software and safety-critical control software.

When a test case has been run once and the output has been confirmed as correct, a common practice is to store that output as a “gold file” for regression testing. When the same test case is run on a new version of the software, the new output is checked to see if it is the same as the “gold” output. However, it may be valid that the new output is different. This can be the case, for instance, if the relative timing of distributed events changes slightly.

Finally, modern computer systems are often subjected to random testing, stress testing or load testing. In such cases, input and system behaviour may not be completely predictable, and the volume of output may be high.

In all of the cases mentioned above, test result checking must typically be more complex than a simple equality check of output against stored output. The phrase “test oracle” is usually reserved for programs that do such more complex analysis of output.

2.2. Why Log Files?

Although test oracles are useful, it is often difficult to capture software input and output directly, and difficult to extract relevant information from captured I/O. The use of text log files addresses these difficulties. Text log files are already in wide use in industry, where they are sometimes referred to as “debug files” or “debug logs”.

Modern software has many diverse inputs and outputs, including mouse input, graphical output, and network and file I/O. This can be difficult to monitor directly unless the software is launched within a platform-specific OS-level sandbox that intercepts all I/O. If instead the SUT itself logs relevant information to a text file, inputs and outputs of diverse devices can be recorded indirectly.

The volume and complexity of I/O can cause problems for direct I/O capture as well. If only some aspects of correctness are to be checked by a test oracle, it may be that only a small part of the actual I/O of a system is needed for checking. Directly captured I/O, such as TCP/IP output of a program, may need to be reparsed and re-interpreted to see whether given high-level events to be checked for have happened. If, instead, selected high-level events are logged to a text file, a smaller amount of focused, easily-parsed information is available.

2.3. Why State-Based Log File Analyzers?

We refer to a test oracle that processes only log files as a *log file analyzer*. Log file analyzers can be written in any programming language, but we have come to believe that languages based on the state machine formalism are the best fit for the task, for three main reasons.

First, we observed that log file analyzers often had to store information about past events in order to detect conformance violations when future events happen. This was sometimes information about which of several discrete states the SUT was in, and sometimes more complex information about numeric and string values appearing in log file lines. This suggests a programming language based on state machines, although the need to also store more complex information suggests that something more than simple finite state machines (FSMs) is needed.

Second, we observed that log files often contain many interleaved streams of information about the SUT, but that checking any one requirement typically involved only a subset of this information. This suggests a programming language in which the checking of separate requirements is assigned to separate state machines.

Third, the state machine formalism is widely-known and used in software engineering in other contexts, such as UML state activity diagrams. The extensions needed for storing more complex information and specifying more than one machine are not major.

2.4. LFAL

We have developed a simple domain-specific language called LFAL (Log File Analysis Language) for writing log file analyzers [5]. LFAL is based on the state machine formalism; however, an LFAL analyzer is *not* an FSM, but rather a collection of (infinite-)state machines running in parallel, in which the states can be any first order terms [6], and in which each machine makes transitions from state to state based on first order terms representing complete log file lines. Conditions on source states and triggering log file lines can be placed on transitions.

LFAL analyzers assume that each log file line starts with a keyword and continues with any number of keywords, strings, integers and real numbers, separated by spaces. Each analyzer machine typically checks conformance to one SUT requirement or a group of related requirements, and notices only a subset of the lines in the log file. If an analyzer machine notices a log file line but has no valid transition on it, it reports an error. We write the state machines for a given analyzer so that this happens if and only if the log file being analyzed shows that a requirement has been violated. (An example of an LFAL analyzer will be developed in Section 3.)

2.5. Related Work

The relation of log file analysis to other work in distributed systems debugging, formal specification, test oracle generation, and assertions is explored in detail in [5]. Because our focus here is on the process by which an LFA test oracle is developed from requirements, we compare this work to similar work in development of formal specifications.

Many papers have dealt with the issue of deriving formal specifications from informal requirements. The target formal specification technologies have included tabular notations [16], the Z specification language [11], the SCR specification methodology [12], and statecharts [9]. These works share a general pattern of describing how to move from informal requirements systematically to the notation or technology in question. In this paper, we follow a similar pattern, concentrating on state machines and providing more detail about intermediate steps required. In addition, the artifact that we end with (the log file analyzer) can be viewed as a formal specification, but is also a program that can be compiled and run for the purpose of test result checking.

Work has been done on deriving requirements and oracles from traces produced by automatic instrumentation [8], although as yet the requirements produced are relatively simple. Some criteria have also been stated for inserting logging instrumentation for the specific purpose of performance profiling [14].

Finally, Cleanroom and other processes based on it [15, 17] share with the processes described here the practice of generating a traceable sequence of artifacts of increasing formality from requirements. In Cleanroom, however, the final artifact is the code itself, and here we are concerned only with a test oracle, which may represent only some of the requirements.

3. Big-Step Process

In this section, we describe a process for deriving logging instrumentation and state-based test oracles from requirements. This process is a distillation of practices that we have followed on previous projects. We call this process the *big-step* process because it involves users taking bigger steps of inference between artifacts than in the small-step process to be described later.

We first describe an example we will use in this paper for expository purposes. We then describe the central artifact of the big-step process, the SPFEs (Situations with Permitted and Forbidden Events), and then go on to describe the process as a whole. We then report on experiences we have had with using and teaching the big-step process, and point out some issues that we have with it.

3.1. Example Software and Requirements

The example software that we will use in this paper for expository purposes is a hypothetical controller for an elevator. We assume that the controller controls both the doors and the motion of the elevator, and we consider the following two requirements.

- R1. The doors are never open when the elevator is in motion.
- R2. Under normal conditions, the elevator door never stays open more than 30 seconds.

The phrase "under normal conditions" in requirement R2 is deliberately vague; we will use it to illustrate how the big-step and small-step processes handle uncertainty in requirements.

3.2. SPFEs

Figure 1 summarizes the big-step process. The central artifact of the process is a list of *Situations with Permitted and Forbidden Events (SPFEs)*. The SPFEs form a link between the language of the requirements and the concepts of state machines.

Each SPFE consists of a situation that the software or its environment may be in, a possibly empty list of events that are permitted in that situation, and a possibly empty list of events that are forbidden in that situation. We use the word "situation" here instead of "state" to avoid confusion with the concept of state-machine states, although we expect that situations in the SPFEs will have a close correspondence with states in the log file analyzer.

SPFEs are best illustrated with some examples. For requirement R1 listed above, a possible set of SPFEs is as follows.

- SPFE1.
  - Situation S1: The elevator door is open.
  - Permitted event P1.1: The door closes.
  - Forbidden event F1.2: The elevator starts moving.
- SPFE2.
  - Situation S2: The elevator is moving.
  - Permitted event P2.1: The elevator stops moving.
  - Forbidden event F2.2: The door opens.
- SPFE3.
  - Situation S3: The elevator is stopped and the door is closed.
  - Permitted event P3.1: The elevator starts moving.
  - Permitted event P3.2: The door opens.

Requirement R2 listed above can be captured by a single SPFE.

- SPFE4.
  - Situation S4: The door last opened at time T_1 and is still open.
  - Permitted event P4.1: The door closes at time T_2, where $T_2 - T_1 \leq 30$.
  - Forbidden event P4.2: The door closes at time T_2, where $T_2 - T_1 > 30$.

Note that in writing the above SPFE, we have implicitly assumed that we are going to use the log file analyzer only when we test the SUT under the "normal conditions" mentioned in requirement R2. This assumption is made more explicit and traceable under the more detailed process described in Section 4.

3.3. Process

The flow of information and the sequence of artifacts produced in the big-step process follows the general pattern of Figure 1. The steps of the process are:

1. From the requirements of the system, derive the SPFEs. The SPFEs should not contradict the requirements, although they can represent only a subset of the requirements if the LFA testing is not intended to cover all requirements.
2. Based on the SPFEs, write a logging policy (LP). The logging policy should specify what events the source code should log and how it should log them. This should include:

Abbr	Expansion
Reqs	Requirements
SPFEs	Situations with Permitted and Forbidden Events
LP	Logging Policy
LFAP	Log File Analyzer Program

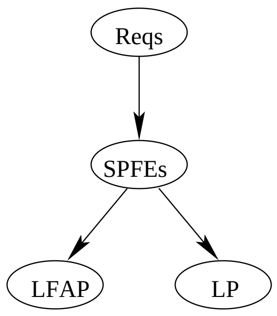


Figure 1. Big-step process summary. Left: Artifact abbreviations and their expansions. Right: Information flow. An arrow indicates that the arrow source is a primary source of information for the arrow destination.

(a) All events that will allow us to determine, for each SPFE, whether we are in the described situation;

(b) All events that are mentioned as "permitted" or "forbidden" in any SPFE.

3. Instrument the source code consistent with the LP.

4. Based on the SPFEs and the logging instrumentation, write and validate the log file analyzer.

In step 1, we do not require every possible event relevant to an SPFE to be listed as either permitted or forbidden, although such a requirement would make the SPFEs more precise. We do this in order to make this step less constrained. If there are particular events that are clearly permitted or clearly forbidden in given situations, then they can be listed as such in the SPFEs, in order to give guidance during steps 2-4. The other events can temporarily be left with their permittedness undefined. Whether they are permitted or forbidden can then be decided when the analyzer is written in step 4.

We now expand upon steps 2, 3 and 4 above. Step 2 requires us to write a logging policy, and step 3 requires us to implement this policy. In our example, if we were considering only SPFE2, we would only need to determine whether the elevator is currently moving or not (S2), whether the elevator has stopped moving (P2.1), and whether the door has opened (F2.2). A sufficient set of events to be logged would be:

- Events in which the elevator starts moving. (Needed to determine whether we are in S2.)
- Events in which the elevator stops moving. (Needed to determine whether we are in S2, and whether P2.1 has occurred.)
- Events in which the door opens. (Needed to determine whether F2.2 has occurred.)

However, for all of SPFE1-4, more events are needed and more data is needed about events in which the door opens. The following set of events is sufficient:

- Events in which the elevator starts moving. (Needed for S2, S3, F1.2, P3.1.)
- Events in which the elevator stops moving. (Needed for S2, S3, P2.1.)
- Events in which the door opens, together with the time the door opens. (Needed for S1, S3, S4, F2.2, P3.2.)
- Events in which the door closes, together with the time the door closes. (Needed for S1, S3, P1.1, P4.1, P4.2.)

The logging policy should state explicitly what format the given events should be logged in. For example, we could say that the SUT must record the above events by logging lines of the form `start_move`, `stop`, `door_open` *t*, and `door_close` *t* respectively, where *t* is a timestamp.

Step 4 of the big-step process requires us to write a log file analyzer. Generally, each analyzer machine is likely to correspond to one or a group of SPFEs, with the Situations corresponding to states of the machine. For example, it is possible to check all of SPFE1-3 with a single LFAL state machine:

```
machine door_safety;
  initial_state closed_stopped;
  from closed_stopped, on start_move,
    to moving;
  from moving, on stop,
    to closed_stopped;
  from closed_stopped, on door_open(T),
    to open;
  from open, on door_close(T),
    to closed_stopped;
  final_state Any.
```

The three states of this machine (`open`, `moving`, and `closed_stopped`) correspond to the Situations in SPFE1-3 respectively. As an example of the treatment of forbidden events, there is no transition on the log file line `door_close` *t* from the state `open`, because that log file line corresponds to an event which is Forbidden by SPFE1; however, there is a transition on the log file line `door_open` *t*, which is a Permitted event. It is possible to check SPFE4 with one additional LFAL state machine:

```
machine door_close_timing;
  initial_state closed;
  from closed, on door_open(T1),
    to open(T1);
  from open(T1), on door_close(T2),
    if (T2-T1 =< 30), to closed;
  final_state Any.
```

Note that the state pattern `open(T1)` contains not only the information that the door is open, but also the time at which the door was opened as a parameter of the state.

3.4. Experiences

We used early, informal versions of the big-step process for several previous projects [4, 2, 13]. The largest analyzer developed and validated was based on the first eight pages of Abrial's Steam Boiler specification [1], and was 333 net lines of LFAL code, containing 19 state machines having a total of 141 transitions. Based on our experiences with those projects, we codified the process for training purposes.

In [10], we reported on an exploratory study of learning and initial use of LFA testing and LFAL. The process for developing log file analyzers which we taught to the learners in this study was the big-step process. We did not monitor how closely they actually followed the process in the study. However, we found that they performed well at the tasks of creating a logging policy, instrumenting code with logging instrumentation, and writing an LFAL log file analyzer.

3.5. Issues

Several issues have come up during our use and teaching of the big-step process that we feel are not handled well by the process. One issue is that it contains no recommendations for what to do when not all the requirements will be checked by LFA testing. We expect that LFA testing will often be used as a complement to traditional testing; that is, not to test all the requirements of a system using LFA, but only a subset, and those only under given conditions. With the big-step process, the decisions made in this regard (e.g. the decision to test the example system only "under normal conditions") are nowhere explicit.

A related issue is lack of documentation and traceability. There is a lack of documentation of the requirements to be checked, the conditions under which LFA testing will take place, and why given events were chosen to be logged. This makes it more difficult to validate big-step process artifacts, e.g. in artifact inspection sessions [7].

Finally, we have noted that in some cases it is difficult to find places in the SUT code at which to log the events needed for the SPFEs. In some cases, the events needed are in a sense "abstract", not able to be matched directly to locations in the code but rather indicating a general pattern of things that have occurred. In these cases we have found that the logging policy is difficult to match up with the SPFEs, the instrumentation in the source code, or both.

These issues are not major if the number of events to be logged or the number of requirements to be checked is small, or in cases of non-critical software. We expect them to be more significant as the number of events or requirements grows and the SUT is more safety-critical.

4. Small-Step Methodology

In this section we propose a process for deriving state machine oracles and logging instrumentation that avoids some of the pitfalls of the big-step process. Because it takes smaller steps toward the production of the final artifacts, we refer to it in this paper as the *small-step* process. It is derived from practices that we have used and observed in the past for making the big-step process more manageable.

We expect the small-step process to be more useful the more the following conditions hold in the development project in question.

- The project is safety-critical, and we want high assurance that requirements are reflected in the test oracle.
- Only a subset of the requirements is to be tested using LFA, and/or LFA testing is only to be done under certain conditions.
- The number of events we need to log is high.
- The requirements to be checked are complex.
- The developers are unfamiliar with using log file analysis for conformance testing, and want to follow a detailed process so that they can learn it.

Figure 2 summarizes the small-step process. The process is similar to the big-step process, but with some important additions for the sake of traceability and generality:

- *RCs (Requirements to be Checked)*: the subset of the original requirements that is to be checked with the LFA testing.
- *CAs (Checking Assumptions)*: the assumptions under which the LFA testing will take place.
- *AEs (Abstract Events)*: all the events that will allow us to tell whether or not we are in the Situations in the SPFEs, and whether or not the Permitted and Forbidden events of the SPFEs have happened.

Abbr	Expansion
RCs	Requirements to be Checked
CAs	Checking Assumptions
SPFEs	Situations with Permitted and Forbidden Events
AEs	Abstract Events
CEs	Concrete Events
LP	Logging Policy
LFAP	Log File Analyzer Program

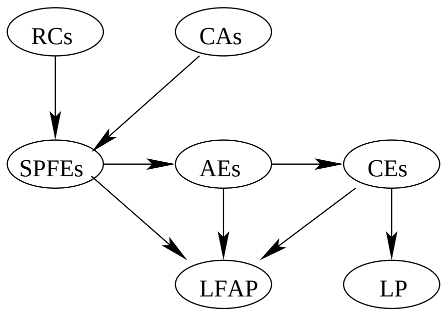


Figure 2. Small-step process summary. Left: Artifact abbreviations and their expansions. Right: Information flow. An arrow indicates that the arrow source is a primary source of information for the arrow destination.

- *CEs (Concrete Events)*: events that are expected to be easily identifiable and loggable at the source code level, that allow us to tell whether or not any of the Abstract Events (AEs) have taken place.

4.1. Small-Step Process Artifacts

We now discuss the major differences between the artifacts generated by the small-step and big-step processes.

RCs (Requirements to be Checked) and CAs (Checking Assumptions). The RCs and the CAs together allow us to inspect the SPFEs for completeness, not just correctness, in cases where LFA testing is used for only some requirements and/or under some testing conditions. The SPFEs should restate all the information in the RCs, given the CAs.

For the elevator controller example, we might state as checking assumptions that we will perform LFA testing of the elevator controller only under the condition that the door is never obstructed by some object, and that this is what "under normal conditions" means in requirement R2. The SPFEs chosen above under the big-step process can then be justified by pointing out that they encompass all the requirements to be checked, given the checking assumptions we have made.

The small-step process is therefore more traceable than the big-step process. Each RC should correspond to one or more SPFEs and vice-versa, and missing details or subcases not considered in the SPFEs should be able to be justified by appealing to the CAs.

AEs (Abstract Events), CEs (Concrete Events), and LP (Logging Policy). AEs are "requirements-level" events, in the sense that their descriptions should be close to the language and terminology of the SPFEs. In contrast, the CEs are "code-level" events, in the sense that it should be possible to match each with one or more locations in the SUT source code at which they should be logged. The AEs and CEs are not necessarily disjoint; there may be some events that are both sufficiently abstract to give information about the SPFEs and also sufficiently concrete to be logged. Such events would be listed as both AEs and CEs.

For our example, we might consider the doors to be open the moment that a command is sent to the door lock actuator to release the door from the closed position. In this case it would be appropriate to take "door open at time t" as an AE, and take "door lock actuator release command at time t" as the CE that will be logged.

Because the CEs are explicitly documented in the small-step process, the logging policy (LP) in the small-step process needs only to document how and when each CE will be logged and the format in which it will be logged. For example, there may be a paragraph in the LP which states:

- When a RELEASE command is sent to the door lock actuator, the line `door_lock_release_cmd` t must be logged, where t is the current timestamp in seconds obtained from `gettimeofday()`.

Again, the small-step process has the advantage of greater traceability over the big-step process. Each paragraph in the LP has to do with one or more CEs; each CE is chosen to allow us to tell whether one or more of the AEs has taken place; and each AE is chosen to allow us to check specific situations and events in the SPFEs.

LFAP (Log File Analyzer Program). The analyzer written in the small-step process is similar to that written in the big-step process, with one exception. We expect the analyzer to contain state machines of two main types: event transducers and SPFE checkers.

Event transducers are relatively simple machines that read CEs from the log file and "convert" them into AEs. In LFAL, we can write state machines that do this by specifying on transitions that the machine "sends" messages standing for the AEs to all the other machines. For instance, a simple single-state machine that translates the "door lock release command" CE into the "door open" AE might be written as follows in LFAL:

```
machine transducer_door_open;
  initial_state null;
  from null, on door_lock_release_cmd(T1),
    to null, sending door_open(T1);
  final_state null.
```

SPFE checkers are machines that check one or more of the requirements, as expressed in the SPFEs. The LFAL state machines described for the big-step process in Section 3.3 are SPFE checkers, and the machines given there would still be appropriate under the small-step process.

4.2. Process

The small-step process starts by generating each of the artifacts mentioned in the table at the right of Figure 2 in that order. Any or all of these steps may cause change requests to be filed against previously-generated artifacts. In particular, we expect the creation of the SPFEs to cause change requests against the RCs and CAs, the creation of the AEs and CEs to cause change requests against the SPFEs, and the creation of the LP and LFAP to cause change requests against all of the SPFEs, AEs, and CEs.

We believe that a thorough development process should include document inspections [7]. Each artifact should be inspected for consistency with upstream artifacts when first created. If numerous change requests have been made against artifacts since the last time they were inspected, a maximally careful process would conduct further inspections to ensure that all artifacts remain consistent with each other. To inspect and validate the LFAP, we have developed tools and techniques for animating the state machines in the analyzer, to help inspectors check that the analyzer is not too strict or lenient [3].

5. Potential Benefits and Problems

Every new software engineering technology brings with it potential benefits and potential problems. Here we enumerate what we see as the most important of both for LFA testing.

5.1. Potential Benefits

Improved reliability. LFA testing assists direct, automatic confirmation of the conformance of software to requirements. This automatic test result checking can be done in complement to traditional human checking and regression checking of test results. The log file analyzer can be written in a language such as LFAL designed for the purpose, and as such can take an abstract, concise view of the requirements.

Note also that, in contrast to other formal methods-based techniques for increasing reliability, no assumptions are made as to the development language or platform of the SUT. As long as the SUT can write a text file, and that file can be transported to a platform on which it can be analyzed, LFA can be used in testing.

Flexibility. LFA testing can be used to check for either a small or a large number of requirements. An elevator controller, for instance, may be a large and complex system having many requirements; however, if only the two example requirements are to be checked by LFA testing, we have shown (Sections 3 and 4) that this can be done by logging only four classes of events and writing as little as 22 lines of LFAL code.

A developer or development team that is unsure about whether LFA testing is appropriate can therefore use it for part of a testing effort, and later evaluate to what extent they wish to use it in the future.

Scalability. Our experience suggests that the number of SPFEs generated by either the big-step or the small-step process is linear in the number of requirements to be checked, and that the size of the final log file analyzer program is linear in the number of SPFEs. We therefore believe that as the number of requirements to be checked by LFA testing increases, the amount of effort to implement LFA testing scales up reasonably.

Traceability. Particularly when using the small-step process proposed here, the log file analyzer program (LFAP) and the logging instrumentation can be traced back directly to the requirements. This traceability can aid in achieving high assurance, in situations in which it is required, for instance in safety-critical projects.

5.2. Potential Problems

False negatives and positives. When an analyzer announces that a log file indicates a fault in the SUT, the cause might actually be a fault in the analyzer. This "false negative" is the equivalent of an invalid expected result in regression testing. A more serious problem is that a faulty analyzer could give a "false positive" by passing a log file that does actually indicate an error.

To address this problem, we have advocated the use of inspections for log file analyzers, and have provided tools for validating them [3]. Of course, the use of these techniques and tools increases the weight of the development process (see below).

Instrumentation maintenance. The logging instrumentation added to the SUT for LFA testing is extra code that must be maintained. Changes to the code for other maintenance reasons must take account of the logging instrumentation.

This problem exists already in the large body of software that generates log files. However, existing log files are often used only for debugging, and are not an intimate part of the test result checking effort; the logging instrumentation therefore does not have to be kept in step with the rest of the code as strictly as with LFA testing.

Process "weight". The benefits that we get out of LFA testing increase as we follow one of the processes outlined in this paper more closely. However, naturally this makes the process more heavyweight and brings with it problems such as developer frustration and process overhead. The more heavyweight the LFA testing process, the more safety-critical the project would have to be to justify it.

6. Conclusions

Whether the benefits of LFA testing outweigh the problems, and under what conditions, are subjects for future research. One of the purposes of this paper is to propose processes for future study that can enhance the benefits and address the problems. We plan to continue by applying the new small-step process to publically-available and industrial requirements, measuring the amount of time taken, the size of the resulting artifacts, and the effectiveness of the resulting testing.

7. Acknowledgements

This research has been generously supported by the Natural Science and Engineering Research Council of Canada (NSERC) and Nokia Corporation. This paper was written while the author was visiting Dan Hoffman and the Computer Science Department at the University of Victoria, to whom we are grateful for the use of their facilities.

References

[1] J.-R. Abrial. Steam-boiler control specification problem. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, volume 1165 of *LNCS*. Springer, October 1996.

[2] X. An. Steam-boiler control system – simulation and testing using log file analysis. Master's thesis, Department of Computer Science, University of Western Ontario, London, Ontario, Canada, September 2000.

[3] J. H. Andrews, R. Fu, and V. D. Liu. Adding value to formal test oracles. In *Proceedings of the 17th Annual International Conference on Automated Software Engineering (ASE 2002)*, pages 275–278, Edinburgh, Scotland, September 2002.

[4] J. H. Andrews and Y. Zhang. Broad-spectrum studies of log file analysis. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 105–114, Limerick, Ireland, June 2000.

[5] J. H. Andrews and Y. Zhang. General test result checking with log file analysis. *IEEE Transactions on Software Engineering*, 29(7):634–648, July 2003.

[6] G. S. Boolos and R. C. Jeffrey. *Computability and Logic*. Cambridge University Press, Cambridge, UK, 2 edition, 1980.

[7] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, July 1976.

[8] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *Proceedings of the 25th International Conference on Software Engineering*, pages 60–73, Portland, Oregon, May 2003.

[9] D. Harel. From play-in scenarios to code: An achievable dream. *IEEE Computer*, 34(1):53–60, January 2001.

[10] G. Huang and J. H. Andrews. Learning and initial use of a software testing technology: An exploratory study. In *Proceedings of the 8th International Conference on Empirical Assessment in Software Engineering (EASE 2004)*, Edinburgh, Scotland, May 2004. To appear.

[11] J. Jacky. *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press, 1997.

[12] J. Kirby, Jr., M. Archer, and C. Heitmeyer. SCR: A practical approach to building a high assurance COMSEC system. In *15th Annual Computer Security Applications Conference*, pages 109–118, Phoenix, Arizona, December 1999.

[13] V. D. Liu. Conformance analysis of communications protocol software using log files. Master's thesis, Department of Computer Science, University of Western Ontario, London, Ontario, Canada, April 2002.

[14] E. Metz and R. Lencevicius. Efficient instrumentation for performance profiling. In *Proceedings of the Workshop on Dynamic Analysis, ICSE 2003*, pages 10–12, Portland, Oregon, May 2003.

[15] H. D. Mills, M. Dyer, and R. Linger. Cleanroom software engineering. *IEEE Software*, 4(5):19–24, September 1987.

[16] D. L. Parnas and P. C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12:251–257, February 1986.

[17] B. D. Tackett and B. V. Doren. Process control for error-free software: A software success story. *IEEE Software*, 16(3):24–29, May/June 1999.

[18] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, November 1982.

Towards Defining and Exploiting Similarities in Web Application Use Cases through User Session Analysis

Sreedevi Sampath
CIS
University of Delaware
Newark, DE 19716
sampath@cis.udel.edu

Amie L. Souter
Computer Science
Drexel University
Philadelphia, PA 19104
souter@cs.drexel.edu

Lori Pollock
CIS
University of Delaware
Newark, DE 19716
pollock@cis.udel.edu

Abstract

With the highly increased use of the web comes a significant demand to provide more reliable web applications. By learning more about the usage and dynamic behavior of these applications, we believe that some software development and maintenance tools can be designed with increased cost-effectiveness. In this paper, we describe our work in analyzing user session data. Particularly, the main contributions of this paper are the analysis of user session data with concept analysis, an experimental study of user session data analysis with two different types of web software, and an application of user session analysis to scalable test case generation for web applications. In addition to fruitful experimental results, the techniques and metrics themselves provide insight into future approaches to analyzing the dynamic behavior of web applications.

1. Introduction

Broadly defined, a web-based software system consists of a set of web pages and components that interact to form a system which executes using web server(s), network, HTTP, and a browser, and in which user input (navigation and data input) affects the state of the system. A web page can be either static, in which case the content is fixed, or dynamic, such that its contents may depend on user input. Dynamic analysis of web applications can provide information that is useful in many ways. For instance, monitoring of an application is used to provide information about the load of traffic of user requests on an application at different times of the day. Knowledge of the pages accessed by individual users is used to customize a web application for more personalization. Information about the dynamic behavior of the application under normal usage can be used for modeling the application for analysis, coupled with modeling based on static information. Logging of the dynamic behavior of a web application can be used for automatic test case generation [8, 21, 28].

In this paper, we describe our work in analyzing user session data. By making minimal configuration changes to a web server, data can be collected as a set of user sessions, each session being a sequence of URL and name-value pairs[1]. The collection of logged user sessions can be viewed as a set of use cases where a use case is a behaviorally related sequence of events performed by the user through a dialogue with the system [11]. By learning more about the usage and dynamic behavior of web applications through user session data analysis, we believe that some software development and maintenance tools can be designed with increased cost-effectiveness. Particularly, the main contributions of this paper are the analysis of user session data with concept analysis, discovering the commonality of URL subsequences of objects clustered in concepts, an experimental study of user session data analysis with two different types of web applications, and an application of user session analysis to scalable test case generation for web applications. In addition to fruitful experimental results, the techniques and metrics themselves provide insight into future approaches to analyzing the dynamic behavior of web applications through analysis of user session data.

2. Clustering via Concept Analysis

Concept analysis is a sound mathematical technique for clustering objects that have common discrete attributes[3]. Concept analysis takes as input a set O of objects, a set A of attributes, and a binary relation $R \subseteq O \times A$, called a *context*, which relates the objects to their attributes. To analyze user sessions using concept analysis we define the objects to represent user sessions, and the attributes of objects are represented by URLs. While a user session is considered to

[1]The name-value pairs are associated with GET/POST requests.

	GDef	GReg	GLog	PLog	GShop	GBooks	GMyInfo
us1	X	X	X				
us2	X	X	X		X		X
us3	X	X	X	X	X		
us4	X	X	X		X	X	
us5	X	X	X				
us6	X	X	X	X	X	X	

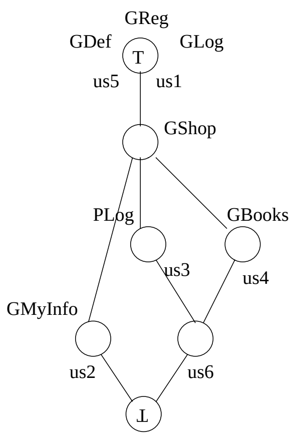


Figure 1. (a) Relation table and (b) concept lattices for test suite reduction

be a set of URLs and associated name-value pairs usually, we currently define a user session during concept analysis to be the set of URLs requested by the user, without the name-value pairs, and without any ordering on the URLs. This problem simplification considerably reduces the number of attributes to be analyzed, and results from our analysis of user sessions described in section 5 provide evidence to justify this simplification.

The relation table in Figure 1(a) shows the context for a set of user sessions for a portion of a bookstore web application [10] which we use for our experiments. Consider the row for the user, us3. The (true) marks in the relation table indicate that user us3 requested the URLs GDef, GReg, GLog, PLog and GShop. We distinguish a GET (G) request from a POST (P) request when building the lattice, since they are essentially different requests.

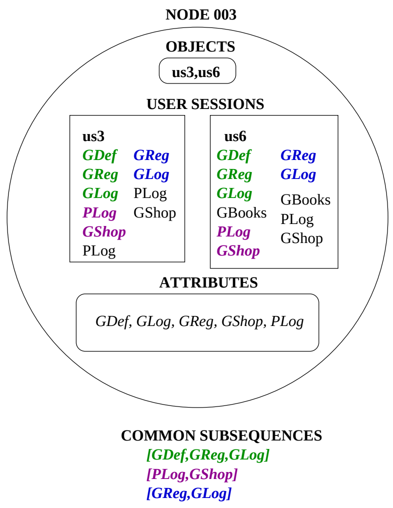
Concept analysis mutually intersects the user sessions for all observed use cases of the web application. The resulting intersections create a hierarchical clustering of the user sessions. Concept analysis identifies all of the concepts for a given tuple (O, A, R), where a *concept* is a tuple $t = (O_i, A_j)$ for which all and only objects in O_i share all and only the attributes in A_j The concepts form a partial order defined as $(O_1, A_1) \leq (O_2, A_2)$, *iff* $O_1 \subseteq O_2$. The set of all concepts of a context and the partial ordering form a complete lattice, called the *concept lattice*, which can be represented by a directed acyclic graph with a node for each concept and edges denoting the $\leq$ partial ordering. Based on the original relation table, concept analysis derives the lattice in Figure 1(b) as a sparse representation of the concepts.

A user session *s* requests all URLs at or above the concept uniquely labeled by *s* in the lattice. Similarly, a URL *u* is accessed by all user sessions at or below the concept uniquely labeled by *u*. The $\top$ of the lattice denotes the URLs that are requested by all the user sessions. The $\perp$ of the lattice denotes the user sessions that access all URLs in the context.

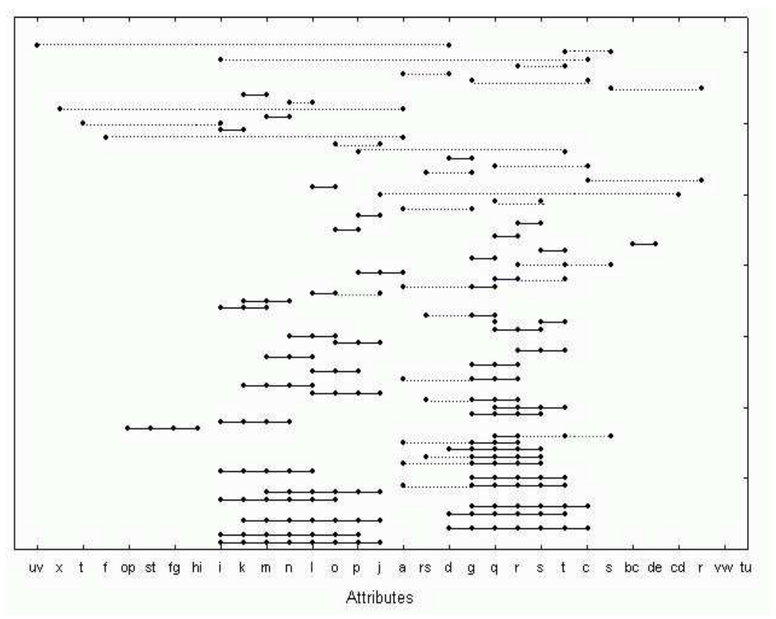
3. Examining Common Subsequences

Clustering of user sessions via concept analysis ensures that all objects in a node have the same set of common attributes. One question arises from clustering based on URL sets without considering the ordering of the URLs in each user session. Will user sessions represented by objects in the same concept node represent similar use cases? To answer this question, a measure of commonality of objects in terms of sequencing (i.e., ordering) of URLs is needed. We propose examining common subsequences as representative of partial use cases of the user sessions.

Figure 2(a) shows an example concept node, the attributes of that node, and the two user sessions represented by the objects of that node. The sequence of URLs for a user session are presented in columns, from left to right. The set of subsequences common to both of these user sessions are indicated below the node. For each concept node containing more than one object, we determine the longest common subsequence (LCS) of URLs among its objects. For different values of *k*, the set of unique subsequences of URLs of length *k* that are common to all the objects in the node is computed. This set is unique, in the sense, that occurrence of the subsequence *[PLog,GShop]* multiple times between the set of objects, is considered only once in the common subsequence set of size 2. Also, if a subsequence *[GDef,GReg,GLog]* is identified as common between objects of the node, then obviously all subsequences of *[GDef,GReg,GLog]* are also common. For the sake of fairness, we do not count subsequences of a larger sequence as smaller subsequences. However, for example, if the sequence *[GReg,GLog]* shows up in our results in addition to *[GDef,GReg,GLog]*, it is because *[GReg,GLog]* occurs as



(a)



(b)

Figure 2. (a) Example of common subsequences and (b) Spread of common subsequences over attributes for a node in Bookstore

a totally different subsequence from the larger subsequence in the use cases of the objects.

Another useful analysis is to examine the spread of the common subsequences of URLs of the objects of a concept node over the attribute space of that node. The graph in Figure 2(b) shows the spread of common subsequences over the attribute space of a node in the lattice for one of the applications we used. The x-axis shows the attributes of the concept node. For ease in showing URL ordering some of the attributes are repeated along the x-axis. This node has 37 attributes all of which are not shown on the axis (because they do not appear in any subsequence of size greater than 1). Continuous subsequences are represented by solid lines. A dotted line between two points, denotes that only the points form the subsequence. Only subsequences of size greater than one are shown in the graph. In this example graph, medium size subsequences cover some of the attributes and other attributes are covered by smaller subsequences. This spread of attribute coverage by common subsequences helps to provide some sense of the overall commonality of the use cases represented by different objects put into the same concept node based only on common sets of URLs.

In section 5.3, our experiments provide evidence that concept analysis with single URLs as attributes clusters objects together such that they have both the same set of attributes and large common partial use cases. In the next section, we describe how clustering these user sessions can be used in scalable test case generation.

4. Application to Test Case Generation

User session based testing exploits the ability of a web server to log user sessions for automatic test case generation. Our key insight to obtaining a scalable approach is to formulate user session based test case generation in terms of concept analysis. Existing incremental concept analysis techniques [9] can be exploited to analyze the user sessions on the fly, and continually minimize the number of maintained user sessions.

In our initial work, we developed a heuristic for selecting a subset of user sessions to be maintained as the current test suite, based on the current concept lattice. Given a context with a set of user sessions as objects O, we define the *similarity* of a set of user sessions $O_i \subseteq O$ as the number of attributes shared by all of the user sessions in O_i. Based on the partial ordering reflected in the concept lattice, user sessions labeling nodes closer to $\perp$ are more similar in their set of URL requests than nodes higher in the concept lattice.

Our heuristic for user session selection, which we call *test-all-exec-URLs*, seeks to identify the smallest set of user sessions that will still cover all of the URLs executed by the original test suite while representing the common URL subsequences of the different use cases represented by the original test suite. This heuristic is implemented as follows: The reduced test suite is set to contain a user session from each node next to $\perp$, that is one level up the lattice from $\perp$. We call these nodes *next-to-bottom* nodes. These nodes contain objects that are highly *similar* to each other. If the set of user sessions at $\perp$ is nonempty, those user sessions are also included. In our example in Figure 1, the original test suite is all the user sessions in the original context. The reduced test suite however contains only user sessions us2 and us6, which label the *next-to-bottom* nodes. By traversing the concept lattice to $\top$ along all paths from these nodes, we will find that the set of URLs accessed by these two user sessions are exactly the set of all URLs requested by the original test suite.

5. Experiments

In order to investigate the effectiveness and usefulness of user session clustering, and our heuristic for user session selection, we performed experiments utilizing a medium and large size application with real user sessions.

5.1. Research Questions

The experiments are designed to answer two questions with regard to user session clustering and selection for scalable test case generation: (1) How effective is the choice of using single URLs as attributes for clustering and is it reasonable to choose only one object from a concept node as the representative object? (2) How effective is the *test-all-exec-URLs* heuristic for selecting test cases for the current test suite? Our hypotheses with regard to these questions are:

1. The set of user sessions (i.e., objects) clustered into the same concept node will have a high commonality in the subsequences of URLs in their sessions. Thus, cost-effective clustering based on single URLs is reasonable, and only one representative from the next-to-bottom nodes can be chosen to be included in the current test suite.
2. In addition to covering all of the executed URLs of the original test suite, the user sessions (i.e., objects) of the next-to-bottom nodes (i.e., in the reduced test suite) execute a high percentage of the subsequences of URLs of the rest of the original test suite. We believe that this provides evidence that the original use cases are well represented by the reduced test suite.

5.2. General Methodology

We use an application from an open source e-commerce site [10] to experiment with applying concept analysis to user sessions to generate a reduced test suite. The application is a bookstore, where users can register, login, browse for books, search for specific books giving a keyword, rate the books, buy books by adding them to the shopping cart, modify personal information, and logout. The bookstore application has 9,748 lines of code, 385 methods and 11 classes. Since our interest was in user sessions, we considered only the code available to the user when computing these metrics, not the code that forms part of bookstore administration. The application uses JSP for its front-end and MySql database for the backend. The application was hosted on the Resin web server[22].

Emails were sent to various local newsgroups, and advertisements were posted in the university's classifieds webpage, asking for volunteers to browse the bookstore. We collected 123 user sessions, all of which were used in these experiments. Some of the URLs of bookstore mapped directly to the 11 classes/JSP files and the rest were requests for gif and jpeg images of the application. The size of the largest user session in bookstore was 863 URLs and on average a user session had 166 URLs.

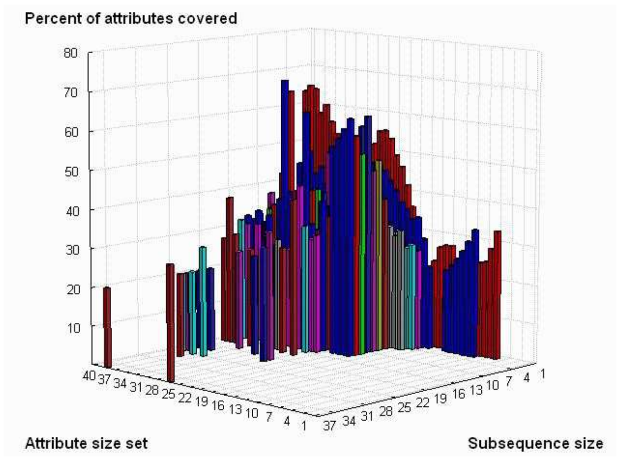
In addition to the bookstore, we also obtained user logs from a University of Delaware production application, uPortal[27], which is an abridged and customized version of the university's web presence, and has options for users to personalize the view of the campus web. The application is mainly open source and is written using Java, XML, JSP and J2EE. uPortal consists of 38,589 lines of code, 4233 methods, and 508 classes. The logs contained 2083 user sessions, which were also analyzed for the experimental study[2]. URLs collected for uPortal mapped directly to 6 of the JSP/Java files, but the data carried on them varied highly for each request. The size of the largest user session in uPortal was 407 URLs and on average a user session had 14 URLs.

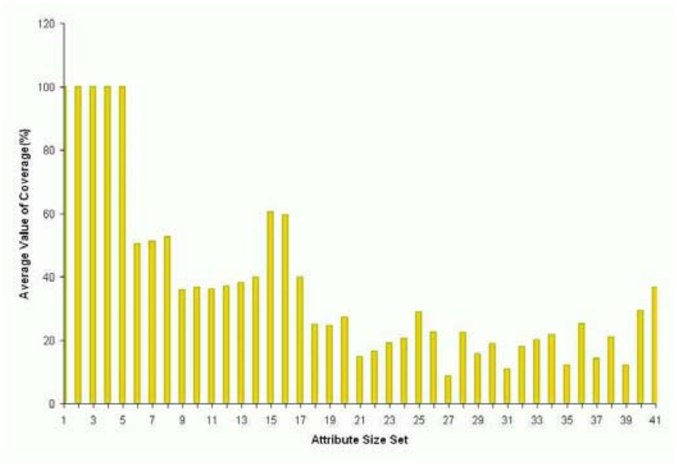
5.3. Commonality Among Attributes of a Concept

The focus of this experiment is to determine if objects clustered together in a concept node, in addition to having a set of common attributes, have a high commonality in the subsequences of URLs. If this is true, then only one object needs to be chosen from a given node for test case generation. Section 3 provided an introduction to the computation of common subsequences of a set of user sessions and our motivation to examine them. This section describes a new metric and experiments we performed towards quantifying the common subsequences of sets of user sessions.

Once common subsequences are generated as described in section 3, the nodes are grouped such that all nodes with the same number *n* of attributes are members of the attr-

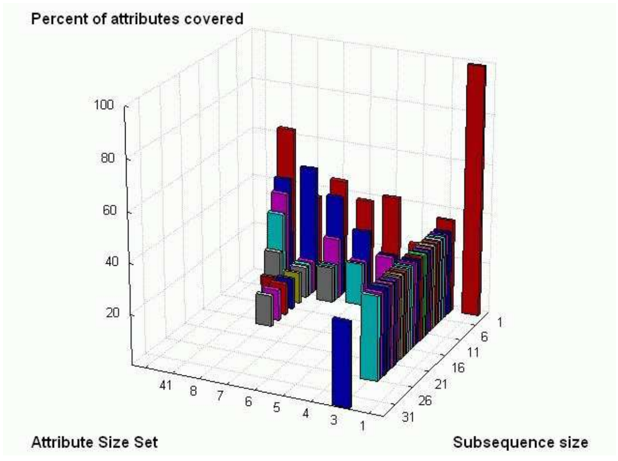
[2]We are currently creating a nonproduction uPortal version that maintains security of individual users' personal information for replay.

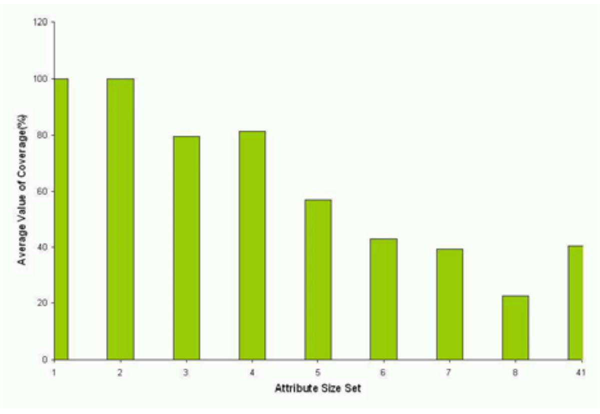




(a) (b)

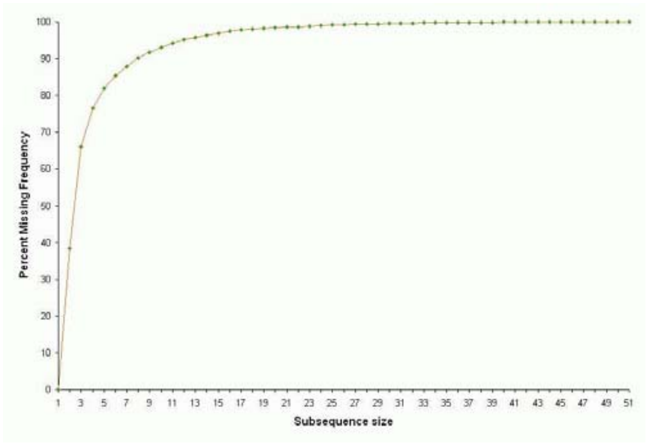
Figure 3. (a) Percent of attributes covered by different subsequence sizes and (b) Average percent of attributes covered by nodes in different attribute size sets for Bookstore

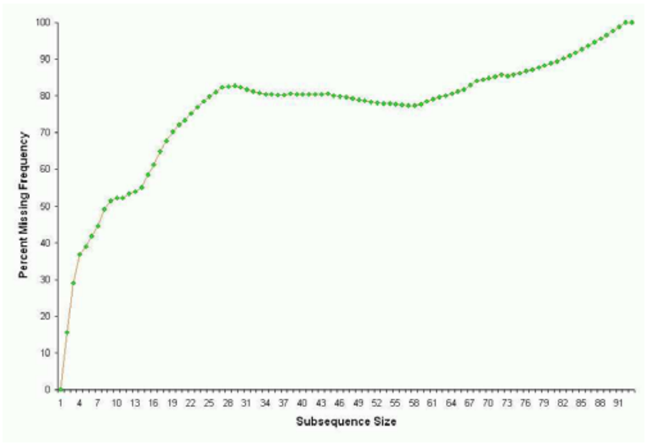




(a) (b)

Figure 4. (a) Percent of attributes covered by different subsequence sizes and (b) Average percent of attributes covered by nodes in different attribute size sets for uPortal





(a) **(b)**

Figure 5. Percent subsequences not covered by *next-to-bottom* nodes for (a) Bookstore and (b) uPortal

size[n] set. This grouping gives a sense of the nodes' level in the concept lattice. Nodes with a low number of attributes will tend to be closer to $\top$.

We define the following metric: the percent of attributes (i.e., URLs) of a concept node that are included in (i.e., covered by) URL sequences of length *k* by objects (i.e., user sessions) in the node. For each attr-size set, the percent of attributes covered by each size subsequence is averaged over all the nodes in the set.

The results of computing this metric for the user sessions collected from bookstore and uPortal are shown in Figure 3(a) and Figure 4(a) respectively. The graphs show the percent of attributes covered by different subsequence sizes of nodes belonging to various attr-size sets. Coverage for subsequences of size 1 is not shown, because it is obviously 100%. As the attr-size increases, the percent coverage of attributes increases. These graphs are simply meant to demonstrate the trends of attribute coverage and attempt to illustrate that subsequences of varying sizes cover a reasonable percent of the attributes. For example, in Figure 3(a), size 10 subsequences across all attr-size sets cover between 27 to 62% of the attributes. In the bookstore, the largest subsequence, size 37, covers 21% of the attributes in one attr-size set and 30% in another. The computation produced similar results with the logs of uPortal (Figure 4(a)) – Size 6 subsequences across all attr-size sets covered between 13 to 33% of the attributes. The largest subsequence of size 33 covers 33% of the attributes.

To enable viewing the trend of percent attributes covered by nodes in different attr-size sets, the results were compiled in a different manner. First, the average percent of attributes covered (a_i) by each concept node *i*, over all subsequence sizes was computed. To be fair, the maximum longest common subsequence value for all nodes in a certain attr-size set is determined and is used in the above average, instead of averaging over the maximum size subsequence of each node. Then, an average percent coverage of the averages (a_i) for all the nodes in an attr-size set was computed. The results are shown in Figure 3(b) for bookstore and Figure 4(b) for uPortal. These graphs demonstrate that the average percent of attributes covered by nodes in various attribute size sets is quite high.

These results strengthen our first hypothesis that indeed there exists commonality in orderings of URLs between objects of a concept node and that these common subsequences cover a high range of percentage of the attributes of that node. Thus, clustering based on single URLs is reasonable for clustering similar use cases, and choosing one object from a given concept node as the representative test case will not result in loss of the attributes covered or the use cases represented by other objects in the node (Question 1).

5.4. Next-to-bottom Coverage of URL Orderings

The goal of this experiment is to support our hypothesis of choosing *next-to-bottom* nodes as the reduced test suite. We believe that such a selection will not cause large loss in representation of use cases associated with the remaining nodes in the lattice.

The *reduced set* is defined to contain the set of objects in the set of *next-to-bottom* nodes of the concept lattice. The difference between the objects that belong to the original test suite and the objects that belong to *reduced set* is called the *remaining set*. This experiment focused on determining the frequency of sequences of URLs that were present in the *remaining set* but missing in the *reduced set*. This metric is our measure to capture the 'loss of coverage' of use cases in the remaining set by the reduced set.

As can be observed for bookstore, in Figure 5(a), for subsequences of size 2, 38.37% of subsequences are missing, 66% of only size 3 subsequences are missing and 76.66% of just size 4 subsequences are missing. For uPortal user sessions, results shown in Figure 5 (b), 15.6% of size 2 subsequences are missing, 29.1% of only size 3 and 36.9% of only size 4 are missing. The percent missing subsequences increases with the size of the subsequence, because it is less likely for two user sessions to share exactly the same long sequence of URLs as many short similar sequences. For uPortal, we observed that there were only a few distinct URLs in the application, and the lengths of the user sessions were relatively small (average of 14 URLs).

It appears that the *reduced set* seems to be lacking the exact same long subsequence present in the *remaining set* but a large number of smaller size subsequences are present. Due to the clustering done by concept analysis, the *reduced set* is guaranteed to have more distinct URLs than the *remaining set*. So even if the exact same long subsequence is absent there are bound to be other sequences (that cover URLs missing in the *remaining set*) that are present in the *reduced set* but absent in the *remaining set*. The absence of a relatively small number of subsequences in the *reduced set* and the assurance due to concept analysis that, a larger number of URLs are present in this set, makes it suitable to be considered for the reduced test suite, and thus moderately supports our second hypothesis (Question 2).

To summarize, the experiments performed in this section support our hypotheses for user session clustering and selection and our heuristic for test case generation. We have performed some preliminary coverage and fault detection studies of test suites created by these techniques and found very promising results. More complete results will be described in a future paper. We believe that the metrics defined and the techniques applied can also be used for other dynamic analysis of web applications.

6. Related Work

Concept Analysis and Clustering in Software Engineering. Snelting first introduced the idea of concept analysis for use in software engineering tasks, specifically for configuration analysis [13]. Concept analysis has also been applied to evaluating class hierarchies [25], debugging temporal specifications [1], redocumentation [14], and recovering components [7, 15, 26, 24]. Ball introduced the use of concept analysis on test coverage data to compute dynamic analogs to static control flow relationships [2]. The binary relation consisted of tests (objects) and program entities (attributes) that a test may cover.

Similar to concept analysis is cluster analysis in which many techniques exist [12]. Such techniques are based on finding groups of clusters in a population of objects, where each object is characterized by a set of attributes. Cluster analysis algorithms use a dissimilarity metric to partition the set of objects into clusters.

To improve the accuracy of software reliability estimation [20], cluster analysis has also been utilized to partition a set of program executions into clusters based on the similarity or dissimilarity of their profiles. It has been experimentally shown that failures often correspond to unusual profiles that are revealed by cluster analysis. Dickinson et al. have utilized different cluster analysis techniques along with a failure pursuit sampling technique to select profiles to reveal failures. They have experimentally shown that such techniques are effective [5, 6]. Clustering has also been used to reverse engineer systems [4, 18, 19, 29].

Test Case Generation. Several tools exists that provide automated testing for web applications such as WebKing [28] and Rational Robot [21]. These tools function by collecting data from users through minimal configuration changes to a web server. The data collected can be viewed as user sessions, which is a a collection of user requests in the form of URL and name-value pairs. To transform a user session into a test case, each logged request of the user session is changed into an HTTP request that can be sent to a web server. A test case consists of a set of HTTP requests that is associated with a particular user session. Different strategies are applied to construct test cases for the collected user sessions. In these tools, test case generators are based on selecting most popular paths in web server logs. Studies have shown promising results that demonstrate the fault detection capabilities and cost-effectiveness of user session-based testing [8]. They showed that the effectiveness of user session techniques improves as the number of collected sessions increases. However, the cost of collecting, analyzing, and storing data will also increase.

Recently, analysis tools have been developed that model the underlying structure and semantics of web-based programs. With the goal of providing automated data flow testing, Liu, Kung, Hsia, and Hsu [16] developed the object-oriented web test model (WATM). They utilized this model to generate test cases, which are based on data flow between objects in the model. Their technique generates def-use chains as test cases, which require additional analysis in order to generate test cases that can be utilized as actual input to the application. They do not indicate how this step would be accomplished.

Ricca and Tonella [23] developed a high level UML-based representation of a web application and described how to perform page, hyperlink, def-use, all-uses, and all-paths testing based on the data dependences computed using the model. Their ReWeb tool loads and analyzes the pages of the application and builds a UML model. The TestWeb tool generates and executes test cases. However, significant intervention is required by the user for generating input.

Lucca et al. [17] recently developed a web application model and set of tools for the evaluation and automation of testing web applications. They developed functional testing techniques based on decision tables, which help in generating effective test cases. However, the process of generating test input in this manner is not automated.

7. Summary and Future Work

This paper has demonstrated that interesting usage patterns of a web application can be uncovered through concept analysis combined with common subsequence analysis. This is just a first step towards better understanding the dynamic behavior of web applications. We have shown how this kind of analysis can be used for scalable, automatic test case generation for this application domain. Our future work includes modifying the heuristic *test-all-exec-URLs* to consider degree of similarity between user sessions, exploring additional user session analyses that might be useful for software engineering tasks, and combining user session analyses with dynamic analysis of the actual program code, towards accurate static modeling of web applications. These combined efforts would provide a basis for creating software development, testing, and maintenance tools for reliable web applications.

Acknowledgements
We thank the U. of Delaware IT group for their aid in logging uPortal sessions.

References

[1] G. Ammons, D. Mandelin, and R. Bodik. Debugging temporal specifications with concept analysis. In *ACM SIGPLAN Conf on Prog Lang Design and Implem*, 2003.
[2] T. Ball. The concept of dynamic analysis. In *ESEC / SIGSOFT FSE*, pages 216–234, 1999.
[3] G. Birkhoff. *Lattice Theory*, volume 5. American Mathematical Soc. Colloquium Publications, 1940.
[4] D. Bojic and D. Velasevic. Reverse engineering of use case realizations in uml. In *Proceedings of the 2000 ACM symposium on Applied computing*, pages 741–747, 2000.
[5] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *Proceedings of the 23rd international conference on Software engineering*, pages 339–348. IEEE Computer Society, 2001.
[6] W. Dickinson, D. Leon, and A. Podgurski. Pursuing failure: the distribution of program failures in a profile space. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 246–255. ACM Press, 2001.
[7] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Trans on Soft Eng*, 29(3):210–224, Mar 2003.
[8] S. Elbaum, S. Karre, and G. Rothermel. Improving web application testing with user session data. In *Int Conf on Soft Eng*, 2003.
[9] R. Godin, R. Missaoui, and H. Alaoui. Incremental concept formation algorithms based on galois (concept) lattices. *Computational Intelligence*, 11(2):246–267, 1995.
[10] Open source web applications with source code. <http://www.gotocode.com>, 2003.
[11] I. Jacobson. The use-case construct in object-oriented software engineering. In J. M. Carroll, editor, *Scenario-based Design: Envisioning Work and Technology in System Development*, 1995.
[12] A. Jain and R. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.
[13] M. Krone and G. Snelting. On the inference of configuration structures from source code. In *Int Conf on Soft Eng*, 1994.
[14] T. Kuipers and L. Moonen. Types and concept analysis for legacy systems. In *Int Workshop on Prog Compr*, 2000.
[15] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Int Conf on Soft Eng*, 1997.
[16] C.-H. Liu, D. C. Kung, and P. Hsia. Object-based data flow testing of web applications. In *Proceedings of the First Asia-Pacific Conference on Quality Software*, 2000.
[17] G. D. Lucca, A. Fasolino, F. Faralli, and U. D. Carlini. Testing web applications. In *International Conference on Software Maintenance*, 2002.
[18] C.-H. Lung. Software architecture recovery and restructuring through clustering techniques. In *Proceedings of the third international workshop on Software architecture*, pages 101–104, 1998.
[19] B. S. Mitchell, S. Mancoridis, and M. Traverso. Search based reverse engineering. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 431–438, 2002.
[20] A. Podgurski, W. Masri, Y. McCleese, F. G. Wolff, and C. Yang. Estimation of software reliability by stratified sampling. *ACM Trans. Softw. Eng. Methodol.*, 8(3):263–283, 1999.
[21] Rational Robot. <http://www-306.ibm.com/software/awdtools/tester/robot/>, 2003.
[22] Caucho resin. http://www.caucho.com/resin/, 2002.
[23] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Proceedings of the International Conference on Software Engineering*, May 2001.
[24] M. Siff and T. Reps. Identifying modules via concept analysis. In *International Conf on Software Maintenance*, 1997.
[25] G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. In *SIGSOFT FSE*, 1998.
[26] P. Tonella. Concept analysis for module restructuring. *IEEE Trans on Soft Eng*, 27(4):351–363, Apr 2001.
[27] Uportal. <http://www.uportal.org>, 2004.
[28] WebKing. <http://www.parsoft.com>, 2004.
[29] T. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Fourth Working Conference on Reverse Engineering (WCRE '97)*, 1997.

Precise detection of memory leaks

Jonas Maebe Michiel Ronsse Koen De Bosschere

Ghent University, ELIS Department
Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
jmaebe|ronsse|kdb@elis.UGent.be
http://www.elis.UGent.be/diota

Abstract

A memory leak occurs when a program allocates a block of memory, but does not release it after its last use. In case such a block is still referenced by one or more reachable pointers at the end of the execution, fixing the leak is often quite simple as long as it is known where the block was allocated. If, however, all references to the block are overwritten or lost during the program's execution, only knowing the allocation site is not enough in most cases. This paper describes an approach based on dynamic instrumentation and garbage collection techniques, which enables us to also inform the user about where the last reference to a lost memory block was created and where it was lost, without the need for recompilation or relinking.

1 Introduction

A memory leak is a memory management problem which indicates a failure to release a previously allocated memory block. The term can be used in two contexts. The first is when indicating imperfections in garbage collectors as used in e.g. Java Virtual Machines, in case they missed the fact that a previously allocated block is not referenced anymore and thus is not added to the pool of free blocks.

The second context is when the programmer himself is responsible for explicitly freeing all blocks of memory that he allocated. This is still the case in most run time environments today and also the situation which we will focus on in this paper.

Leaking blocks of memory during a program execution has several negative consequences. It often results in said program acquiring more and more memory from the operating system during its execution.

As such, overall system performance will degrade over time, as allocated but unused blocks of memory will have to be swapped out once the system runs out of free physical memory. Eventually, a program may even exhaust its available virtual address space, which will cause it to terminate due to an out-of-memory error.

Several packages that can perform memory leak detection already exist. The necessary instrumentation can happen at different levels. Insure++ [5] rewrites the source code of an application. Many leak detectors operate at the library level by intercepting calls to memory management routines, such as in case of LeakTracer [1], memdebug, memprof and the Boehm Garbage Collector [2].

Finally, it is possible to instrument at the machine code level. Purify [8] statically instruments the object code of an application and the libraries it uses. Dynamic instrumentors such as Valgrind [7] delay the instrumentation until run time.

Except for Insure++, all of the mentioned debugging helpers only tell the programmer where the leaked block of memory was allocated, but not where it was lost. Insure++ does show where the last pointer to a block of memory was lost, but not where this pointer got its value. Additionally, since it is a source code instrumentation tool, it requires recompilation and cannot provide detailed information about leaks in third-party libraries of which the source code is unavailable.

In this paper, we present a technique that uses dynamic instrumentation at the machine code level to track all pointers to allocated blocks of memory. It is completely language- and compiler-independent and can show where the leaked blocks were allocated, lost and where the last references to these blocks were created.

In what follows, we first give a short overview of the instrumentation framework we use. Next, we discuss the kinds of memory leaks that exist and how they may occur. We then describe in great detail how we can detect these leaks, as well as some implementation details. Finally, we conclude after presenting a short evaluation and discussing our future plans.

2 Instrumentation overview

The inner workings of the instrumentation framework that we use, DIOTA (which stands for Dynamic Instrumentation, Optimization and Transformation of Applications), are explained extensively in [6]. The framework itself is quite generic, and specific instrumentation applications are realised through so-called backends. These are shared libraries that link to the DIOTA-framework and which specify how DIOTA should rewrite the code during the instrumentation.

The techniques we will describe rely on only two features of DIOTA: the ability to intercept calls to dynamically linked routines and being notified of memory operations. The former enables us to track the memory allocations and deallocations performed by the program, the latter is used to track the pointers to the memory blocks as they are passed through the program.

3 Memory leaks

There are two kinds of memory leaks. ZeroFault Software [4] calls them logical and physical. A logical memory leak occurs when a block of memory is allocated and never freed afterwards, but at all times during the program execution a reachable pointer to this block of memory exists. A physical memory leak occurs when the last reachable pointer to a particular block of memory is lost.

We mainly focus on physical memory leaks in this article, because finding out where exactly the last pointer to a block of memory is lost is crucial to fix such an error and this information is often hard to come by. The described techniques are however also applicable to solving logical memory leaks. In that case, at the end of the program our technique allows us to provide the developer with a list of references to all unfreed memory blocks, including the place where they were created.

A physical memory leak can occur in three ways:

- The last reference to a block of memory is overwritten with a new value, or some value is added to it. In the latter case, it is possible that the original value will be restored later by subtracting this same value again, so one should take this into account to avoid false positives.
- The last reference to a block of memory goes out of scope. For example, it was stored in a local variable or a parameter and the function exits.
- The block of memory containing the last reference to another block of memory is freed.

Note that the lost reference to a block of memory does not really have to be the last one for it to cause a physical memory leak. In case of cyclic structures, it is possible to be left with a group of blocks all referring each other, but no way to reach them anymore from global data pointers or local variables.

In order to discover a physical memory leak, a way to track all pointers to a particular block of memory is required. In this sense, the problem is identical to the classic problem of garbage collection. One can therefore also choose from the wide variety of known algorithms to perform garbage collection in order to find memory leaks.

There is however one important difference as far as finding physical memory leaks is concerned: one wants to know as exactly as possible where a block of memory was lost. Periodic garbage collection can only loosely pinpoint where the last reference to a block disappeared, and more exact techniques are required to improve accuracy.

For this reason, we have chosen to use reference counting [9] as opposed to e.g. the more commonly used mark-and-sweep algorithm. Although this increases the overhead significantly and prevents us from detecting leaked cycles, we think that the added detailed information is worth it. Additionally, it is still possible to periodically perform a mark-and-sweep to detect leaked cycles.

4 Detection

4.1 Memory blocks

In order to be able to track pointers to allocated memory blocks, one first has to know where those blocks are located. For this purpose, our DIOTA-backend intercepts all calls to malloc, calloc, realloc, free, new, delete and their variants.

In case of allocations, the replacements call through to the original functions and then record their return values. This recording occurs in a hash table, with the blocks hashed on their start address. Since we only count the references to the start address of a block, this allows for enough flexibility as far as searching is concerned. The deallocation routines remove the block to be deallocated from this hash table before actually deallocating it.

For each allocated block, quite a bit of information is recorded. First of all, the call stack at the time of allocation is stored. Next, we also give each allocated block a reference count, a unique identifier (called a *block id*), and a *usecount*. This last field keeps track of how many times a reference to said block has already been created and will allow us to detect stale references as explained in the next section.

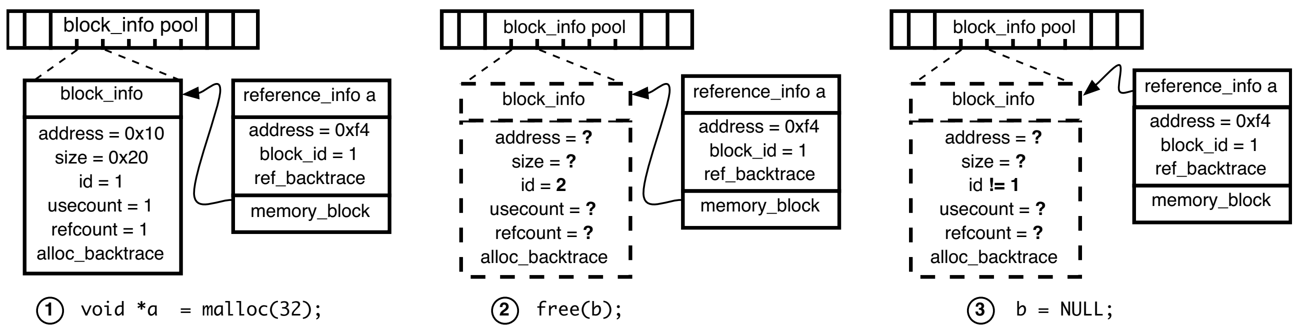


Figure 1. Reference and block bookkeeping

4.2 References

The second piece of the puzzle is keeping track of all references to these blocks of memory. For each reference we keep track of where it was created, its address, the memory block it refers to and the *block id* of the block when the reference was created.

The information about these references is stored in structures residing in two trees, with one tree reserved for the references residing on the stack. A first reason to separate the stack items from the rest, is that measurements showed that many more references are created and removed on the stack than elsewhere, and at the same time the maximum number of references located on the stack at a single time is often a factor 1000 smaller than the maximum of references residing on the heap.

It thus makes sense to keep the volatile but small group of references on the stack separate from the rest for performance reasons. Additionally, when the stack shrinks, we can keep removing the first item of the stack tree as long as this item's address lies below the new stack pointer, simplifying stack shrinking management considerably.

A final useful property of these trees is that when a memory region is copied (e.g. using `memcpy`), we can easily find all references lying inside this region in order to copy them as well, without having to scan the entire copied region or having to iterate over all recorded references.

The bookkeeping of the references can be achieved by looking at the results of all store operations performed by the program. Load operations are largely irrelevant, as most of the time they only result in an extra reference when the value is stored back to memory. Register variables can be handled by looking at the contents of the registers when the reference count of a block drops to zero.

When a value equal to the start address of an allocated block is stored, we increase the reference count of said block. When a previously recorded reference is overwritten with a different value, the reference count of the block it referred to is decreased again.

After a block has been freed however, all of its references become stale. There are two ways to solve the problem of stale references: one is to find (or keep track of) and remove all those references, another is to make sure the staleness can be detected the next time this reference is accessed. We use the latter technique to avoid the extra associated with the former.

The staleness detection is implemented using the unique identifier that each block possesses: as mentioned before, creating a reference to a block results in the current *block id* of that block to be copied to the reference's information structure. When a memory block is freed, its *block id* is set to the next available unique value. As such, when we afterwards encounter a stale reference to this block, we can immediately notice this due to the fact that the *block ids* do not match.

This technique also allows us to immediately make a structure containing the information about a memory block available for reuse (through a dedicated pool of such structures) when its corresponding block is freed. Even though there may be stale references to such a block and thus this structure, the unique identifier makes sure this can be detected reliably when the referencing occurs.

Figure 1 shows an example of how this works in practice. First, the program allocates a block of memory. After calling the real `malloc`, we allocate a memory block info structure from the previously mentioned dedicated pool and fill in the appropriate values. The fact that this info block is then stored in a hashtable, is not shown here.

Still in statement 1, the program stores the pointer to this block in the variable `a`. At this point, we create a new reference info structure. The pointer to the memory block info structure is a pointer in the programming language sense: it is simply the address of this structure.

The block is freed again in statement 2. As shown, the memory block information structure is freed at the same time, but the reference information structures are left intact. The *block id* of the memory block is increased though.

When the program afterwards overwrites `a` with a new value, possibly after new blocks have been allocated, the situation will be as shown for statement 3. The memory

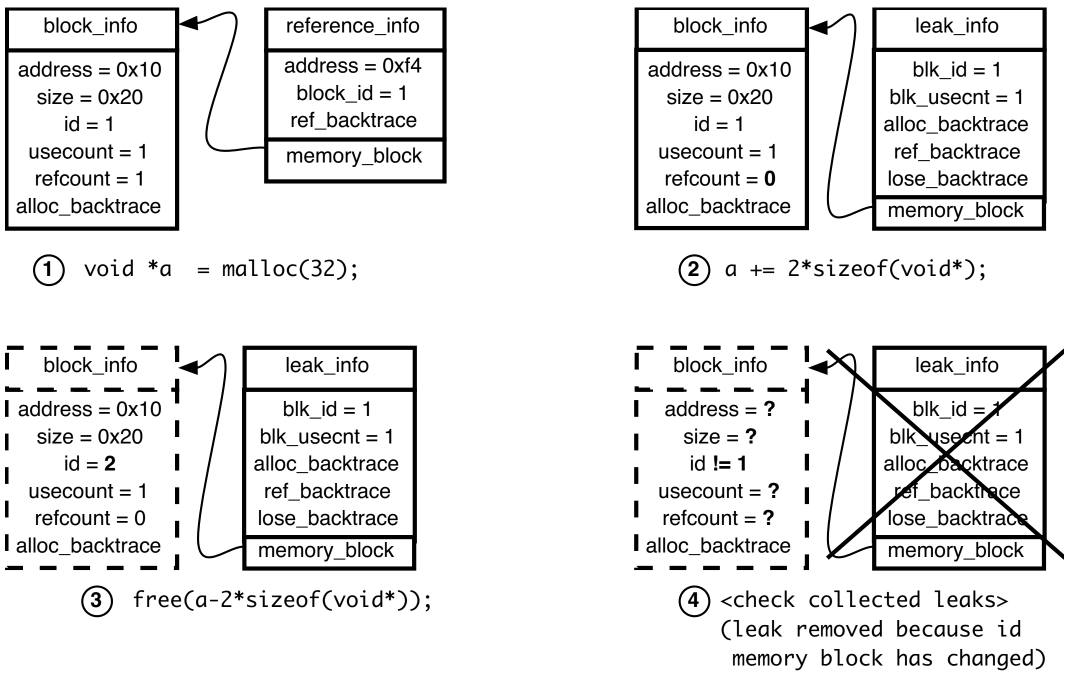


Figure 2. Example of detecting a false positive

block information structure may either be free or in use for another memory block that has been allocated in the mean time, but we can detect the fact that the reference originally pointed to another memory block by comparing the *block id's*.

4.3 Leaks

Finally, there are the memory leaks. When the reference count of a block of memory reaches zero, a new *potential* leak is created. The are called potential leaks because there may still be a reference to the possibly leaked block in a register, or it could be that a new reference will be calculated by the program later on (e.g. by substracting a value from a pointer that currently points to somewhere in the middle of that block).

Such a potential leak contains the call stacks of where it occurred, where the last reference, which was just lost, was created and where the leaked block was allocated.

All potential leaks are stored in a hash table, with the hash based on the recorded call stacks mentioned above. Apart from that data, we also record the cause of the leak (as explained in section 3) and the current *block id* and *usecount* of the memory block at the time the leak occurred. Finally, leaks also have an occurrence count.

Two potential leaks are deemed identical if their causes and their three recorded call stacks match. In such a case, the previously stored potential leak in the hash table is verified to see whether the block it refers to is still leaked.

This verification occurs at two levels. First of all, if the block id recorded in the potential leak is different from the current one of the memory block, it means the block has been freed since we detected the potential leak, so it was a false positive.

The second verification is based on discrepancies between the *usecount* values of the potential leak and the memory block. If these values differ, a new reference has been created to the supposedly leaked block since the original leak detection. As such, this block can not have been leaked at that moment in time.

If both tests pass, the previously recorded leak is deemed to be permanent. The occurrence count of the leak is increased, and the the stored *block id* and *usecount* are replaced by those of the newly detected leak.

A demonstration of detecting a false positive based on a change of *block id* is shown in figure 2. Like in the previous figure, in statement 1 a memory block is allocated and the resulting pointer is stored in variable a. The result is that an info structure for the memory block and the reference are created, with the latter referring to the former.

In statement 2, we add a constant to a. When the value of a in memory is changed, we detect that the new pointer no longer refers to the start of the memory block, so we decrease the reference count of the previous block it pointed to.

Since that one is now zero, we create a new potential leak. This leak gets a copy of the creation backtrace of the reference we just overwrote, the allocation site of the memory block and the current backtrace (i.e., the place where the leak was detected). We also copy the current *block id* and *usecount* values of the memory block and keep a pointer to the information about the memory block for future checking.

In statement 3, the block is freed. Consequently the *block id* of the memory block info is increased. The potential leak remains untouched. Note that if the parameters are passed via the stack, the *usecount* of the block will also be increased, since by passing the parameter a new reference to the block is created.

When we later on check whether the previously created potential leak was a real leak, we can see it was not due to the fact the *block ids* will differ between the memory block info and the leak info (as well as the *usecount*, possibly).

4.4 Reporting

Every time memory is allocated, we check whether 5 minutes have passed since the last time we wrote out all collected leaks. If so, then we process all collected potential leaks, write out the ones we deem to be real leaks (using the same verification based on *block ids* and *usecounts* described in the previous section) and reset the hash table containing them. This procedure is also carried out when the program exits.

We only do this checking at allocation time, since if the program is not allocating any new memory, any leaks that may have happened are not going to have much influence on the program's operation. We also do not lose any information by delaying the reporting of the leaks.

We have not yet implemented the reporting of the remaining references to logically leaked memory blocks at the end of the program. It could be easily done by iterating over all still existing references though, preferably grouping them per leaked memory block.

An example of a report (in verbose mode) of a detected leak can be seen in figure 3

5 False positives, false negatives

A very important aspect of detecting memory leaks, is dealing with false positives and false negatives. In case of a real garbage collection system, one cannot afford to incur the former, as it would result in memory corruption. In our case, the consequences are not as catastrophic, but if there are too much false positives, the output becomes useless to the user.

We use the system of the potential leaks to avoid most false positives. The majority of those result from functions which return the last reference to a block of memory in a register. Once the stack shrinks, the last reference is then often removed, resulting in a potential leak. When the result of this function is stored back to memory, the *usecount* of the memory block is increased, so the false positive will be recognised and not reported.

Another way to deal with this, would be to scan the contents of the registers whenever a leak due to stack shrinking occurs, but that has not yet been implemented.

Another kind of false positive can occur since we only track references to the start of a memory block. In practice, we only experienced this in the case of C++ code, where in some cases constructors return a pointer to `sizeof(void*)` bytes past the start of the allocated block. We compensated for this by treating such pointers also as references to blocks. After this adaption, we did not encounter any further reported false positives due to pointers not pointing to the start of a block.

Permanent false positives can occur due to not handling cases which seldom happen, such as overwriting part of a pointer, or writing a pointer byte per byte to memory. Adding support for these cases can be done at the expense of a larger slowdown.

False negatives can occur when a value is stored to memory that happens to have the same value as the start of an allocated memory block, but which is not actually used in the program as such. In case this is a loop counter, the reference count will immediately be decreased again in its next iteration. If it is random data, e.g. copied from a mapped file, a physical leak may never be detected using the reference counting method. The same goes for leaked cyclic structures.

However, since we keep track of all allocated memory blocks, we can still report them as logical memory leaks when the program exits. Additionally, we can also provide a list of all sites where the remaining references were created.

6 Related work

As mentioned in the introduction, several memory debuggers which support memory leak detection already exist. Most simply provide replacement functions for malloc, free and friends and report, when the program exits, which blocks have not been deallocated. This very low overhead technique is used by Valgrind, Leaktracer, memdebug and memprof and is therefore useful to try first.

Insure++ performs full instrumentation of the available source code and can therefore also track where exactly the last reference to a block of memory is lost in case of physical memory leaks, at least if this occurs within a part of the program for which source code is available. It does not provide any extra information regarding logical memory leaks.

Another interesting case is the well known Boehm garbage collector [2]. It includes a mode in which it functions as a memory leak detector instead of as a garbage collector. However, as it relies on periodic scanning of the

```
*** Warning, freed block containing last reference to a block of memory
    (reference at 0x88a937c, block at 0x88a9388, ip = 0x8048497,
     1 occurrence(s)) in thread 0 at:
        [0x08048497]: test_linked_list, /user/jmaebe/diota/test/mem8.c:54
                53:    // make a->next no longer reachable
                54:    free(a);
                55:  }
        [0x08048642]: main, /user/jmaebe/diota/test/mem8.c:120
                119:    testje2(&a);
                120:    test_linked_list();
                121:    // test multiple leaks at the same location
        [0x0039176b]: __libc_start_main+235, /lib/tls/libc.so.6

The last reference to that block we know of was created at 0x804848e:
        [0x0804848e]: test_linked_list, /user/jmaebe/diota/test/mem8.c:51
                50:    a=malloc(sizeof(record_t));
                51:    a->next=malloc(sizeof(record_t));
                52:
        [0x08048642]: main, /user/jmaebe/diota/test/mem8.c:120
                119:    testje2(&a);
                120:    test_linked_list();
                121:    // test multiple leaks at the same location
        [0x0039176b]: __libc_start_main+235, /lib/tls/libc.so.6

This block was allocated at 0x8048486:
        [0x08048486]: test_linked_list, /user/jmaebe/diota/test/mem8.c:51
                50:    a=malloc(sizeof(record_t));
                51:    a->next=malloc(sizeof(record_t));
                52:
        [0x08048642]: main, /user/jmaebe/diota/test/mem8.c:120
                119:    testje2(&a);
                120:    test_linked_list();
                121:    // test multiple leaks at the same location
        [0x0039176b]: __libc_start_main+235, /lib/tls/libc.so.6
```

Figure 3. Example of verbosely reported memory leak

```
...
     enode* result = new_enode(polynomial, exp+1, pos+1/*from 1 to m*/);
     for(int i=0;i<exp+1;i++) {
        set<map<lstring,int> > new_terms =
          find_terms_with_var_exp(terms, var_name, i);
        // fix memory leak found by DIOTA
        value_clear(result->arr[i].d);
        value_clear(result->arr[i].x.n);
        result->arr[i] = translate_one_term(parameter_names,
                                            left_over_var_names,
                                            new_terms);
     }
...
```

Figure 4. Bug found in FPT using our technique

address space of a program using a variant of the mark-and-sweep algorithm, it can only discover that a pointer got lost somewhere between two garbage collections.

7 Evaluation

We evaluated our techniques by analysing a few known free software programs (lynx and vim, which turned out not to contain any recurring memory leaks), as well as locally adapted versions of the SimpleScalar simulator and the Fortran Parallel Transformer (FPT) [3]. The slowdown factor lies between 200 and 300 times, which is obviously very significant. The amount of required memory more or less doubles compared to the original execution.

Both SimpleScalar and FPT were known to contain memory leaks from testing with other tools, but without the exact location of the actual leaking, fixing them proved to be very hard. An example from FPT is shown in figure 4.

Originally, the `d` and `x.n` fields in lines 6 and 7 were `long int`'s. Afterwards, they were changed into whole numbers with infinite precision from the GNU Multiprecision Library GMP. Before overwriting such values, one has to call the `value_clear()` macro to free previously allocated memory.

While adding such calls throughout 50000+ lines of C++ code, the two that are now at lines 6 and 7 were forgotten. Our tool pinpointed what is now line 8 in the fragment above as the place where the last reference to a block of memory was overwritten.

8 Future plans

One of our main goals currently is to reduce the overhead of our backend. It has already become more than a factor 10 faster since the start of this project, and we are confident we can reduce it a lot more. One way is to adapt DIOTA so that the backend can better control when exactly it wants its callbacks to be called.

Currently, the backend's callback is called before each memory access (either load, modify or store). This means that when a store is reported, the new value is not yet written to memory. As such, we have to log this event and only when the next memory operation occurs, the result of the previous store can be examined.

Another issue is detecting when the stack shrinks. At the moment, every time a memory access occurs, we check whether the stack has shrunk and whether consequently some references went out of scope. A much better way would be to insert these checks only after instructions that can increase the stack pointer (given a downward growing stack).

We therefore intend to add a mode to DIOTA whereby a backend's callbacks will only be called right after a write or modify operation, and add the ability for a backend to specify on a per-instruction (type) basis whether it wants to be called or not.

9 Conclusion

In this paper, we described how precise memory leak detector can be performed using the reference counting technique. We described implementation details and the problems of false positives and false negatives.

We showed in our evaluation that although the current slowdown is quite big, the results provided by the technique help significantly with finding the root cause of memory leaks. We intend to speed up the implementation and technique in the future.

10 Acknowledgements

Jonas Maebe is supported by a grant from the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). This research was also funded by Ghent University and by the Fund for Scientific Research-Flanders (FWO-Flanders).

The authors also wish to thank Kristof Beyls for providing ideas, testing consecutive implementations and his invaluable feedback throughout this process.

References

[1] E. S. Andreasen. Leaktracer. `http://www.andreasen.org/LeakTracer/`.
[2] H. Boehm. Dynamic memory allocation and garbage collection. In *Computers in Physics*, volume 9, pages 297–303, May 1995.
[3] E. D'Hollander, F. Zhang, and Q. Wang. The fortran parallel transformer and its programming environment. *Journal of Information Science*, 106:293–317, 7 1998.
[4] T. Z. Group. Zerofault. `http://www.zerofault.com`.
[5] Insure++. `http://www.parasoft.com/`.
[6] J. Maebe, M. Ronsse, and K. D. Bosschere. DIOTA: Dynamic Instrumentation, Optimization and Transformation of Applications. In *Compendium of Workshops and Tutorials, Held in conjunction with PACT'02: International Conference on Parallel Architectures and Compilation Techniques*, Charlottesville, Virginia, USA, Sept. 2002.
[7] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In O. Sokolsky and M. Viswanathan, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.
[8] E. R. Rs. Purify: Fast detection of memory leaks and access errors. `http://citeseer.nj.nec.com/291378.html`.
[9] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.

On Specifying and Monitoring Epistemic Properties of Distributed Systems

Koushik Sen, Abhay Vardhan, Gul Agha, Grigore Roşu
Department of Computer Science
University of Illinois at Urbana Champaign
{ksen,vardhan,agha,grosu}@cs.uiuc.edu

Abstract

We present an epistemic temporal logic which is suitable for expressing safety requirements of distributed systems and whose formulae can be efficiently monitored at runtime. The monitoring algorithm, whose underlying mechanism is based on symbolic knowledge vectors*, is distributed, decentralized and does not require any messages to be sent solely for monitoring purposes. These important features of our approach make it practical and feasible even in the context of large scale open distributed systems.*

1. Introduction

The discovery and prevention of software errors is a difficult problem involving many different aspects, such as incorrect or incomplete specifications, errors in coding, faults and failures in the hardware, operating system or network. Two prominent formal approaches used in checking for errors are: theorem proving and model checking. Theorem proving is powerful but labor-intensive, requiring intervention by someone with fairly sophisticated mathematical training. On the other hand, model checking is more of a push-button technology, but despite exciting recent advances, the size of systems for which it is feasible remains rather limited. As a result, most system builders continue to rely on testing to identify bugs in their implementation.

There are two problems with software testing. First, testing is generally done in an *ad hoc* manner: it requires the software developer to translate properties into specific checks on the program state. Second, test coverage is rather limited. To mitigate the first problem, software often includes dynamic checks on the systems state to identify problems at run-time. Recently, there has been some interest in run-time monitoring techniques [1] which provide a little more rigor in testing. In this approach, monitors are automatically synthesized from a formal specification. These monitors may then be deployed off-line for debugging or on-line for dynamically checking that safety properties are not being violated during system execution.

In [6] we argue that distributed systems may be effectively monitored against formally specified safety requirements. By effective monitoring we mean not only linear efficiency, but also decentralized monitoring where few or no additional messages need to be passed for monitoring purposes. We introduced an epistemic temporal logic for distributed knowledge, called *past time linear temporal logic* and abbreviated PT-DTL, and showed how monitors can be synthesized for it. PT-DTL formulae are local to particular processes and are interpreted over projections of global state traces that the current process *is aware of*. In this paper, we increase the expressiveness of PT-DTL and make it more programmer friendly by adding constructs similar to value binding in programming languages and quantification in first order logic. These constructs allow us to succinctly specify properties of open distributed systems involving data. The new logic is called XDTL and its novel features are inspired from EAGLE [3].

Let us assume an environment in which a node a may send a message to a node b requesting a certain value. The node b, on receiving the request, computes the value and sends it back to a. There can be many such nodes, any pair can be involved in such a transaction, but suppose that a crucial property to enforce is that no node receives a reply from another node to which it had not issued a request earlier. One can check this global property by having one local monitor on each node, which monitors a single property. For instance, a monitors "if a has received a value from b then it must be the case that previously in the past at b the following held: b has computed the value and at a a request was made for that value in the past". Using XDTL, all one needs to do is to provide the safety policy as a formula:

$$\texttt{valueReceived} \rightarrow \\ @_b(\Diamond(\texttt{valueComputed} \wedge @_a(\Diamond \texttt{valueRequested})))$$

$@$ is an *epistemic operator* and should be read "at"; $@_b F$ is a *remote property* that should be thought of as the value of F in the most recent local state of b that the current process is aware of. In PT-DTL[6], $@$ can only take one process as a subscript. In XDTL, as described later in the paper, $@$ can take any set of processes as a subscript together with a universal or an existential quantifier, so $@_b$ becomes "syntactic sugar" for $@_{\forall\{b\}}$ (or for $@_{\exists\{b\}}$). $\Diamond$ should be read "eventually in the past". Monitoring the formula above will involve sending no additional messages but only a few bits of information piggybacked on the messages already being passed for the computation.

Suppose that we want to restrict the above safety policy by imposing a further condition that the value received by a must be same as the value computed by b. To express this stronger property, we need to compare values in states at two process that are not directly related. This property cannot be directly expressed in PT-DTL without introducing extra variables in the program itself. However, adding extra variables in the program can potentially result in side-effects which are not desirable. An elegant way to solve the problem is to introduce the notion data-binding in the logic used for monitoring. Informally, we can restate the property as follows: a monitors "if a has received a value from b then remember the value received in a variable k and it must be the case that previously in the past at b the following held: b has computed the value and the computed value is equal to k and at a a request was made for that value in the past". This can be written formally as follows:

$$\begin{array}{l}\texttt{valueReceived} \rightarrow \ \mathsf{let}\ k = \texttt{value}\ \mathsf{in} \\ \quad @_b(\Diamond(\texttt{computedValue} \wedge (k = \texttt{valueComputed}) \\ \qquad \wedge @_a(\Diamond\texttt{requestedValue})))\end{array}$$

Informally, the construct "$\mathsf{let}\ \vec{k} = \vec{\xi}\ \mathsf{in}\ F$" binds the value of the expressions $\vec{\xi}$ at process a with the logic variables $\vec{k}$ which can be referred by any expression in the formula F.

Another example in [6] regards monitoring certain correctness requirement in a leader-election algorithm. The key requirement for leader election is that there is at-most one leader. If there are 3 processes namely a, b, c and `state` is a variable in each process that can have values `leader`, `loser`, `candidate`, `sleep`, then we can write the property at every process as: "if a leader is elected then if the current process is a leader then, to its knowledge, none of the other processes is a leader". We can formalize this requirement as the following PT-DTL formula at process a:

$$\begin{array}{l}\texttt{leaderElected} \rightarrow (\texttt{state} = \texttt{leader} \rightarrow \\ \quad (@_b(\texttt{state} \neq \texttt{leader}) \wedge @_c(\texttt{state} \neq \texttt{leader}))\end{array}$$

We can write similar formulae with respect to b and c. Given an implementation of the leader election problem, one can monitor each formula locally, at every process. If violated then clearly the leader election implementation is incorrect.

However, the above formula does not specify the requirement that every process must know the name of the process that has been elected as leader. We cannot express this stronger requirement in PT-DTL. However, using the construct "let _ in _" and assuming that the variable `leaderName` contains the name of the leader, the requirement can easily be stated in XDTL as follows:

$$\begin{array}{l}\texttt{leaderElected} \rightarrow \mathsf{let}\ k = \texttt{leaderName}\ \mathsf{in} \\ \quad (@_b(\texttt{leaderName} = k) \wedge @_c(\texttt{leaderName} = k))\end{array}$$

Note that the above formula assumes that the name of every process involved in leader election is known to us beforehand. Moreover, the size of the formula depends on the number of processes. In a distributed system involving a large number of processes, writing such a large formula may be impractical. The problem becomes even more important in an open distributed system where we may not know the name of processes beforehand. To alleviate this difficulty, as already mentioned, we use a set of indices instead of a single index in the operator @. The set of indices denoting a set of processes can be represented compactly by a predicate on indices. For example, in the above formula, instead of referring to each process by its name we can refer to the set of all remote processes by the predicate $i \neq a$ and use this set as a subscript to the operator @:

$$\begin{array}{l}\texttt{leaderElected} \rightarrow \mathsf{let}\ k = \texttt{leaderName}\ \mathsf{in} \\ \quad @_{\forall\{i|i \neq a\}}(\texttt{leaderName} = k)\end{array}$$

$@_{\forall\{i|i \neq a\}}(\texttt{leaderName} = k)$ denotes the fact that the formula $\texttt{leaderName} = k$ must hold true at all processes i satisfying the predicate $i \neq a$. This is equivalent to the first order logic formula $\forall i\ .\ ((i \neq a) \rightarrow @_i(\texttt{leaderName} = k))$.

The logic XDTL proposed in this paper, extending PT-DTL with the construct "let _ in _" and with quantified sets of processes in the subscript of the epistemic operator @, is more expressive and elegant than PT-DTL. These benefits are attained without sacrificing efficiency and the decentralized nature of monitoring.

Many researchers have proposed temporal logics to reason about distributed systems. Most of these logics are inspired by the classic work of Aumann [2] and Halpern *et al.* [4] on knowledge in distributed systems. Meenakshi *et al.* define a knowledge temporal logic interpreted over a message sequence charts in a distributed system [5] and develop methods for model checking formulae in this logic. However, in our work we address the problem of monitoring and investigate an expressive distributed temporal logic that can be monitored in a decentralized way.

The rest of the paper is organized as follows. Section 2 describes the basic concepts of distributed systems. Section 3 introduces the more expressive PT-DTL which we call XDTL. In Section 4 we conclude by briefly sketching a decentralized monitoring algorithm.

2. Distributed Systems

We consider a distributed system as a collection of processes, each having a unique name and a local state, communicating with each other through asynchronous message exchange. The computation of each process is abstracted out in terms of *events* which can be of three types: *internal*, an event denoting local state update of a process, *send*, an event denoting the sending of a message by a process to another process, and *receive*, an event denoting the reception of a message by a process. Let E_i denote the set of events of process i and let E denote $\bigcup_i E_i$. Also, let $\lessdot \subseteq E \times E$ be defined as follows.

1. $e \lessdot e'$ if e and e' are events of the same process and e happens immediately before e',
2. $e \lessdot e'$ if e is the send event of a message at some process and e' is the corresponding receive event of the message at the recipient process.

The partial order $\prec$ is the transitive closure of the relation $\lessdot$. This partial order captures the *causality* relation among the events in different processes and gives an abstraction of the *distributed computation* denoted by $\mathcal{C} = (E, \prec)$. In what follows, we assume an arbitrary but fixed distributed computation $\mathcal{C}$. Let us define $\preccurlyeq$ as the reflexive and transitive closure of $\lessdot$. In Fig. 1, $e_{11} \lessdot e_{23}$ and therefore also $e_{11} \prec e_{23}$. However, even though $e_{12} \not\lessdot e_{23}$, we have $e_{12} \prec e_{23}$ as process 2 gets a message from process 3 which contains knowledge of e_{12}.

The *local state* of a process is abstracted out in terms of a set of events. For $e \in E$ we define $\downarrow e \overset{\text{def}}{=} \{e' \mid e' \preccurlyeq e\}$, that is, $\downarrow e$ is the set of events that causally precede e. For $e \in E_i$, we can think of $\downarrow e$ as the local state of process i when the event e has just occurred.

We extend the definition of $\lessdot$, $\prec$ and $\preccurlyeq$ to local states such that $\downarrow e \lessdot \downarrow e'$ iff $e \lessdot e'$, $\downarrow e \prec \downarrow e'$ iff $e \prec e'$, and $\downarrow e \preccurlyeq \downarrow e'$ iff $e \preccurlyeq e'$. We use the symbols s_i, s_i', s_i'' and so on to represent the local states of process i. We also assume that each local state s_i of each process i associates values to some local variables V_i, and that $s_i(v)$ denotes the value of a variable $v \in V_i$ in the local state s_i at process i.

We use the notation $causal_j(s_i)$ to refer to the latest state of process j of which process i knows while in state s_i. Formally, $causal_j(s_i) = s_j$ where s_j is a state at process j such that $s_j \preccurlyeq s_i$ and for all states s_j' in process j with $s_j' \preccurlyeq s_i$ we have $s_j' \preccurlyeq s_j$. For example, in Figure 1 $causal_1(\downarrow e_{23}) = \downarrow e_{12}$. Note that if $i = j$ then $causal_j(s_i) = s_i$.

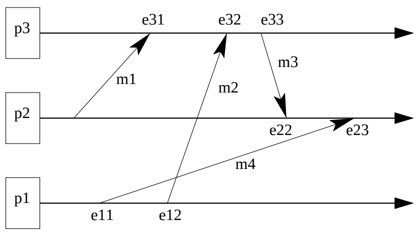


Figure 1. Sample Distributed Computation

3. Extended Distributed Temporal Logic

In order to reason about the global distributed computation locally, XDTL has a set of three new variants of *epistemic operators*, whose role is to evaluate an expression or a formula in the *last known state* of a remote process. We call such an expression or a formula *remote*. In addition to the epistemic operators, we add the construct "let $\vec{k} = \vec{\xi}$ in F" to XDTL to bind expressions to local logic variables that can be referred by any expression or formula in F.

The intuition underlying XDTL is that each process may be associated a local formula which, due to the epistemic operators, can refer to the global state of the distributed system. These formulae are required to be valid at the respective processes during a distributed computation. The distributed computation satisfies the specification when all the local formulae are shown to satisfy the computation. Next, we formally describe the syntax and semantics of XDTL.

3.1. Syntax

In the sequel, whenever we talk about an XDTL formula, it is in the context of a particular process, having the name i. We call such formulae *i-formulae* and let F_i, F_i', etc., denote them. Additionally, we introduce the notion of expressions local to a process i called as *i-expressions* and let ξ_i, ξ_i', etc., denote them. Informally, an i-expression is an expression over the global state of the system that process i is currently aware of. Local predicates on i-expressions form the atomic propositions on which the temporal i-formulae are built.

We add the *epistemic operators* $@_{\forall J} F_j$ and $@_{\exists J} F_j$ which is true if at all (or some, respectively) processes j in the set J, F_j holds. Similarly, we add the epistemic operator $@_J \xi_j$ which returns the set of j-expressions ξ_j for all processes j in the set J. The sets J can be expressed compactly using predicates over j. For example, J can be the sets $\{j \mid j \neq a\}$ or $\{j \mid client(j)\}$. The following gives the formal syntax of XDTL with respect to a process i, where i and j are the name of any process (not necessarily distinct):

$F_i ::=$	true $\mid$ false $\mid P(\vec{\xi_i}) \mid \neg F_i \mid F_i \; op \; F_i$	propositional
	$\mid \odot F_i \mid \Diamond F_i \mid \boxdot F_i \mid F_i \; \mathcal{S} \; F_i$	temporal
	$\mid @_{\forall J} F_j \mid @_{\exists J} F_j$	epistemic
	$\mid$ let $\vec{k} = \vec{\xi_i}$ in F_i	binding
$\xi_i ::=$	$c \mid v_i \mid k \mid f(\vec{\xi_i})$	functional
	$\mid @_J \xi_j$	epistemic
$\vec{\xi_i} ::=$	$(\xi_i, \ldots, \xi_i)$	

The infix operator *op* can be any binary propositional operator such as $\wedge, \vee, \rightarrow, \equiv$. The term $\vec{\xi_i}$ stands for a tuple of expressions on process i. The term $P(\vec{\xi_i})$ is a (computable) predicate over the tuple $\vec{\xi_i}$ and $f(\vec{\xi_i})$ is a (computable) function over the tuple. For example, P can be $<, \leq, >, \geq, =$. Similarly, some examples of f are $+, -, /, *$. Variables v_i belong to the set V_i containing all the local state variables of process i. c stays for constants, e.g., 0, 1, 3.14.

3.2. Semantics

The semantics of XDTL extends the semantics of PT-DTL by defining the three variants of epistemic operators

$$
\begin{array}{ll}
\mathcal{C}, s_i, [e] \models @_{\forall J} F_j & \text{iff } \forall j \,.\, (j \in J) \rightarrow \mathcal{C}, s_j, [e] \models F_j \text{ where } s_j = causal_j(s_i) \\
\mathcal{C}, s_i, [e] \models @_{\exists J} F_j & \text{iff } \exists j \,.\, (j \in J) \wedge \mathcal{C}, s_j, [e] \models F_j \text{ where } s_j = causal_j(s_i) \\
\mathcal{C}, s_i, [e] \models \mathsf{let}\ (k, \ldots, k') & = (\xi_i, \ldots, \xi_i') \ \mathsf{in}\ F_i \quad \text{iff } \mathcal{C}, s_i, [e, k \mapsto (\mathcal{C}, s_i, [e])[\![\xi_i]\!], \ldots, k' \mapsto (\mathcal{C}, s_i, [e])[\![\xi_i']\!]] \models F_i \\
 & \\
(\mathcal{C}, s_i, [e, k \mapsto val])[\![k]\!] & = val \\
(\mathcal{C}, s_i, [e])[\![@_J \xi_j]\!] & = \{(\mathcal{C}, s_j, [e])[\![\xi_j]\!] \mid s_j = causal_j(s_i) \wedge j \in J\}
\end{array}
$$

Table 1. Semantics of XDTL

and the binding operator. The semantics is given by recursively defining the satisfaction relation $\mathcal{C}, s_i, [e] \models F_i$, where $[e]$ is an environment carrying the bindings for different logic variables which gets introduced by the "let _ in _" operator. $(\mathcal{C}, s_i, [e])[\![\xi_i]\!]$ is the value of the expression ξ_i in the state s_i under the environment $[e]$. Table 1 formally gives the semantics of the new operators of XDTL. For the semantics of other operators the readers are referred to [6]. We assume that expressions are properly typed. Typically these types would be `integer, real, strings`, etc. We also assume that $s_i, s_i', s_i'', \ldots$ are states of process i and $s_j, s_j', s_j'', \ldots$ are states of process j.

4. Monitoring Algorithm

To monitor XDTL formulae in a decentralized way, we synthesize *distributed monitors* as follows. For each process there is a separate monitor, called a *local monitor*, which checks the local XDTL formulae and can attach additional information to any outgoing message. This information can subsequently be extracted by the local monitor on the receiving side without changing the underlying semantics of the distributed program. The local monitor of each process i maintains a KNOWLEDGEVECTOR data-structure KV_i, storing for each process j in the system the status of all the safety policy sub-formulae and sub-expressions referring to j that i is aware of. The knowledge vector KV_i is appended to any message sent by i. When performing an internal computation step, the status of the local formulae and expressions is automatically updated in the local knowledge vector. When receiving a message from another process, the knowledge vector is updated if the received message contains more recent knowledge about any process in the system. To do this, a sequence number needs also to be maintained for each process in the knowledge vector. Unlike [6], the entries of KNOWLEDGEVECTOR are symbolic expressions instead of values. This is due to the fact that all the logic variables referred in an expression or a formulae may not be available at the time of evaluation of the expression or the formula. Therefore, the evaluation of a formula or an expression may be partial, containing the various logic variables. The logic variables in these formulae or expressions are replaced by actual values once they become available. A detailed discussion of the algorithm is beyond the scope of this short paper. However, readers are referred to [6, 3] for some of the similar ideas.

5. Conclusion

We believe that the logic XDTL presented in this paper is a powerful underlying specification formalism for distributed systems. Specifications expressed as XDTL formulae can be effectively monitored, even in the context of large scale open distributed systems. However, it is worthwhile to investigate other extensions that increase its expressiveness without sacrificing the efficiency of monitoring.

Acknowledgements

The first three authors are supported in part by the DARPA IPTO TASK Program, contract F30602-00-2-0586, the DARPA IXO NEST Program, contract F33615-01-C-1907, the ONR Grant N00014-02-1-0715, and the Motorola Grant RPS #23 ANT. The last author is supported in part by the joint NSF/NASA grant CCR-0234524.

References

[1] *1st, 2nd and 3rd CAV Workshops on Runtime Verification (RV'01 - RV'03)*, volume 55(2), 70(4), 89(2) of *ENTCS*. Elsevier Science: 2001, 2002.

[2] R. Aumann. Agreeing to disagree. *Annals of Statistics*, 4(6), 1976.

[3] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proceedings of 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *LNCS*.

[4] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning about Knowledge*. MIT Press, 1995.

[5] B. Meenakshi and R. Ramanujam. Reasoning about message passing in finite state environments. In *International Colloquium on Automata, Languages and Programming (ICALP'00)*, volume 1853 of *LNCS*.

[6] K. Sen, A. Vardhan, G. Agha, , and G. Roşu. Efficient decentralized monitoring of safety in distributed systems. In *Proceedings of 26th International Conference on Software Engineering (ICSE'04) (To Appear)*.

Ideas for Efficient Hardware-Assisted Data Breakpoints

Jonathan E. Cook   Mayur Palankar
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003 USA
jcook@cs.nmsu.edu

Abstract

Data breakpoints, sometimes called watchpoints, have long been desirable for debugging and other dynamic analyses, but are often prohibitively slow to use. Current processors have a small number of breakpoint registers that can be used to trap data read and write operations at CPU speeds—for example, the Intel 386+ CPUs have four breakpoint registers that can watch one word of memory each. Current use of these registers is naive and limited, and so we propose and describe some investigation into furthering their use.

1. Introduction

While debuggers have long supported efficient code breakpoints, data breakpoints, sometimes called watchpoints, have lagged behind in the efficiency. This is because debuggers have typically resorted to single-stepping through the program and checking to see if the current instruction is going to touch the watched data location. Code breakpoints are easy because there is only one place in the code to worry about, and a trap can easily be set at that point. A data location can be used or assigned in many code locations, and in a program with pointers it is possibly undecidable as to which code locations will affect a specific data location. One reference cites a slowdown of 85,000 times for a program running under a debugger with a data watchpoint set [3].

Current processors have attempted to alleviate this situation somewhat by including in their design a small number of breakpoint registers that can be used to trap data read and write operations at CPU speeds—for example, the Intel 386+ CPUs have four breakpoint registers that can watch one word of memory each. Other CPUs have just one data breakpoint register.

These registers are a step forward but are obviously a severely limited resource. For example, the *gdb* debugger will use the breakpoint registers for simple variable access breakpoints, but will resort to software trapping if more data is being watched than there are registers, or if expressions are used. For example, the program

```
int main()
{
   int x,y,z;
   y = 5; z = 3;
   for (x=0; x<10000000; x++)
   {
      if (x == 678456)
         y = 2;
      z = x - 5;
   }
   z = (x-z) + y;
   return z;
}
```

when run directly gives an execution time of 0.04 seconds. Running it under *gdb* with no watchpoints gives an instantaneous prompt return (meaning essentially no slowdown), and running it with a "watch y" also gives an instantaneous program interruption at the "y=2" line. Unfortunately, running the program while watching for the expression "y==5" to change takes 6 minutes, 15 seconds in the debugger process and 41 seconds in the program process. This gives a total slowdown of about 10,400.

Thus, while the data breakpoint registers are being used currently, their use is basic and naive. We propose several interesting research questions surrounding the use of these registers. Can these limited breakpoint registers be used efficiently to watch a large number of data locations? If so, what types of extra support are needed to be able to schedule the registers? Can static analysis of the program help determine the scheduling of which locations needs to be watched at each point in the program runtime? If 100% coverage is too expensive to obtain, can statistical methods be used to achieve high but not perfect coverage?

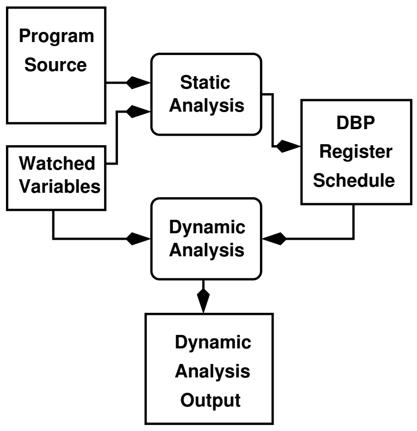


Figure 1. General framework for breakpoint register usage.

In this paper we explore some not-yet-tested thoughts on how these data breakpoint (DBP) registers might be used for efficient dynamic analyses that need to watch variables. Figure 1 shows our general framework for efficiently using the DBP registers in dynamic analyses. We propose that a static analysis phase is needed to build an efficient (or even feasible) schedule when trying to use limited DBP registers to watch a large number of variables.

2. Assumed instrumentation capabilities

Our goal is to improve the efficiency of dynamic analyses that need to watch variables. Thus, while in the extreme we might need full instruction-level instrumentation capabilities, our goal is to use as little instrumentation as possible. The two questions regarding watching more variables than are DBP registers are 1) Does a schedule for the DBP registers exist that covers all accesses to the variables, and 2) What triggers are needed in the program execution to allow us to change the current watched variables according to the schedule? In this section we ignore question 1 for now.

At the initial level, we of course have the DBP registers themselves. We assume that on each trigger of our instrumentation, we can know the current statement in the program. Also note that DBP triggers are not necessarily lightweight. Our current mechanism (and the only known capability) places triggers in a parent process (similar to a debugger); thus a trigger causes a heavyweight context switch to another process.[1]

If the data breakpoints themselves are not sufficient, then what other instrumentation do we need? A first step would be function entry and exit, with a more generalized notion being basic block entry and exit. Having access to the scope entry and exit of variables that are being watched is of obvious benefit. It may also be the case that schedule changes might need to be done within a scope but still at the beginning or end of some intermediate basic block.

The final, most detailed level, would be the ability to instrument arbitrary points in the program, between individual statements and even at the expression level. This would allow schedule changes at any point in the program. At this level our task seems essentially equivalent to register allocation. It is important to remember, however, that modifying the DBP registers is likely to be a heavyweight operation (e.g., involving a context switch), and thus its frequency needs to be minimized. Again, experimental evidence will be needed to decide how often this level of instrumentation will be needed.

It is important to remember that the DBP registers watch for accesses to a memory location. They do *not* attempt to correspond data register accesses to memory location accesses. Thus, as long as variable values are in a register and being used in a register, the variable accesses are invisible to the DBP mechanism. The full implications of this are not yet absorbed by the authors, but it certainly implies that DBP registers do not necessarily provide full trapping of all variable accesses, especially in optimized programs and on architectures with a large number of registers.

3. Watching multiple variables

The first scenario that we consider is the simple case of watching more variables than there are breakpoint registers. In this section we are assuming that the program does not use pointers.

In considering how to watch the variables, the first thing we need to look at is the lifetimes of the variables. If the number of overlapping variable lifetimes is less than or equal to the number of breakpoint registers, then watching them is fairly straightforward. It still requires, however, a schedule for which variables are watched when, and the selection of triggers on which to change the schedule. Because DBP registers watch addresses, it would be incorrect to watch a local variable's address while out of scope for that variable. Thus, even without overlapping lifetimes outnumbering our DBP

[1]Initial measurements over a simple program where a trigger occurred every 10th iteration through a counting loop resulted in a slowdown of about 50—the original unwatched process took 12.41 seconds, while the watched process (and the monitor) took 116.1 user seconds and 502.51 system seconds. While much better than the 10,400 slowdown of the debugger, this is quite high for hardware-assisted breakpoints, and the numbers show that most of the overhead is in the kernel.

registers, scheduling triggers may still need to resort to instrumentation at the scope entry/exit level. It should never need any finer level of instrumentation for this case.

Next is the case where the number of overlapping lifetimes is greater than the number of DBP registers. Here there can be no guaranteed minimal instrumentation level. Yet there still can be hope for the higher levels. If static analysis can determine a set of variable watches that must trigger and that can indicate a point to change one or more DBP registers to watch other variables, then even this case might be handled by the highest level of instrumentation.

We think that the best way to evaluate this is to analyze some benchmark programs over a variety of watch sets and determine what is needed.

4. Introducing pointers

Pointers, as always, are the bane of analyzers. Moreover, pointers are probably the exact cause of many of the problems we might want to find by watching variables. A programmer wants to know exactly when and where a variable is first getting clobbered.

Many studies have been done on points-to analyses, and it is often the case that pointers have very small points-to sets (e.g., [2]). This is encouraging in that it provides hope that the potential program sites for reading and writing watched variables does not greatly increase with pointers in the code. However, with even one pointer having a large or all-variables points-to set (including our watched variables, one of which it may be clobbering!) and a watch set greater than the number of DBP registers, we are immediately in trouble in how to set our DBP registers.

A potential solution is, in these cases, to watch the pointer rather than the variables in our watch set. This allows us, with one DBP register while in the scope of that pointer, to be able to determine if the accesses of the pointer will read or write any of the variables. We can then "virtually" trigger the instrumentation on the variables themselves. This idea can be applied not just at the scoping level of troublesome pointers, but at the statement level. Thus, essentially, they become additional variables to watch, with an even higher priority than our regular variables.

A downside to this approach is that the slowdown of the program might greatly increase due to the interruptions caused by pointer accesses. For non-safe pointers we would potentially need to trap every read of the pointer because it could cause a write to one of our variables. We should be able to eliminate through static analysis the program locations where the pointers are only being accessed to read the data, but this may be infrequent or not beneficial in average programs. We could also skip pointer reads when we know the current value of the pointer is not one of our watched variables, and only trap on the next pointer write (assuming no pointer arithmetic).

5. Static analyses

Since we are considering the problem of watching data accesses, it seems natural that dataflow analyses are the types of analyses that would most directly inform our dynamic analysis stage.

With def-use information and points-to sets from a pointer analysis, we would know where in the CFG does each watched variable need to be watched. If variables are being watched only for writes, then blocks with variable definitions matter. If we are watching reads and writes, then both defs and uses matter.

We assume that we can have a statement or expression-level CFG if needed, rather than simply a basic-block CFG. This would allow us to ensure that there is no CFG node does not indicate more watched variables than are physically possible.

With the above information, the essential problem is that of creating an efficient schedule. By efficient we mean one that is cost-minimally updated. This may not be the same as one with the minimal number of updates. For example, since DBP triggers already interrupt the process when they occur, it would probably be cheaper to update the schedule from within DBP triggers even if we need extra or more updates than with a scheme that needed to add special traps (causing new context switches) to achieve an absolute minimum number of updates.

Attacking this problem might introduce some new analyses that have heretofore not been considered. For example, if we are using write-only DBP triggers, then if a watched variable write is dominated by a set of writes on currently watched variables, then that set can potentially serve as schedule update points to bring in the new variable needing watched. In other words, we need to find the def(X)-def(Y) chains, where a definition of watched variable X is live at the definition of watched variable Y. Finding dominance relations over these chains would give us points for potential schedule changes in the DBP triggers themselves.

Similar inter-variable dataflow analysis would be needed for variables being watched for read and write accesses. While it sounds daunting at first, this type of analysis would only need performed over watched variables (and some pointers), not all variables, and may be an effective (or necessary) way of finding schedule updates based on the DBP triggers themselves.

6. Statistical tracing

For some programs and set of desired variables to watch, it might be the case that 100% coverage of all variable accesses is simply too prohibitive in cost to achieve. This might be because the program has many ill-defined pointers that need to be watched constantly and thus cause many program interrupts, or because the number of watched variables and their interaction is such that DBP register schedule changes need to be made so often that it results in too much instrumentation overhead.

Thus, we may wish to attempt to catch *most* variable accesses, but with much less instrumentation. Rather than consider the whole space a continuum over which to make this tradeoff, for now we simplify the problem to the following question. With DBP triggers and function entry/exit triggers, can we schedule the DBP registers to catch a high percentage of watched variable accesses?

We feel that an empirical investigation into this will be the only way to really answer the question, given the range of possible programs and specifications of watched variables that can be involved. While there will likely be no guaranteeable achieved coverage, perhaps a static analysis phase could optimize the schedule and warn about likely code areas where large numbers of variable accesses may be missed.

A somewhat tangential but related idea is that of saving the previous value of each watched variable at each DBP trigger. This would only require a doubling (plus some overhead) of the watched variable space, and would provide a safeguard mechanism for potentially noting missed writes on watched variables. If the current value at a DBP trigger is the same as the previous, we cannot say for certain that there was no write in between (it might have written the same value), but if it is different we have definitely detected an intervening write that was missed by our DBP triggers.

7. Related work

Wahbe et al. [3] present the closest related work, in which they attack the problem of data breakpoints. Their motivation is the same, and they give an example of a slowdown of 85,000 when data breakpoints are used in a debugger. Interestingly enough, they mention the existence of data breakpoint registers, but do not use them in their work. They dismiss them because of their limited numbers (the Intel i386+, at four, seem to have the most). Rather, they take a code-patching approach, and they do employ some static analysis steps to reduce the amount of instrumentation.

Ball and Larus discuss the optimization of program tracing in [1], but their work is focused on control-flow tracing, and optimal placement of instrumentation to capture enough information to reconstruct the original control flow.

8. Conclusion

Data breakpoint registers, although few, offer hardware support for dynamic analyses that need to observe data accesses. In trying to create efficient instrumentation for dynamic analyses, we should use, as best we can, every resource that is available. To this end, we presented ideas for how the data breakpoint registers might be used and managed to watch a large number of variables.

Our ideas center around performing some static analysis in order to determine a schedule of DBP allocation that will cover the variable accesses we are interested in. Some harder issues that we have not yet thought about are multithreaded programs with global variables, shared memory pages between processes, and other mechanisms that step outside of the bounds of single-thread access to data.

On the practical side, it is interesting to note that the only implementation support for using data breakpoint registers is highly inefficient, forcing a context switch to a monitoring (parent) process. While this may be natural for user-controlled debuggers to use, automatic runtime monitors would benefit from new, efficient support for these hardware resources.

Acknowledgments

This work was supported in part by the National Science Foundation under grants CCR-0306457, EIA-9810732, and EIA-0220590. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

[1] T. Ball and J. Larus. Optimally Profiling and Tracing Programs. 16(4):1319–1360, July 1994.

[2] M. Hind and A. Pioli. Which Pointer Analysis Should I Use. In *Proc. 2000 International Symposium on Software Testing and Analysis*, Aug. 2000.

[3] R. WAhbe, S. Lucco, and S. Graham. Practical Data Breakpoints: Design and Implementation. In *Proc. 1993 Conference on Programming Language Design and Implementation*, pages 1–12, June 1993.

SAAT: Reverse Engineering for Performance Analysis

Seon-Ah Lee, Seung-Mo Cho, Sung-Kwan Heo
Software Center, Corporate Technology Operations, Samsung Electronics Co. Ltd.
599-4, Shinsa-dong, Kangnam-gu,Seoul, Korea, 135-120
{salee, seungm.cho, sk.heo}@samsung.com

Abstract

It is essential to understand both the static and dynamic aspects of existing software for performance analysis. Software reverse engineering reestablishes the structure and behavior of software and helps with that understanding. Researchers in reverse engineering, however, have focused on identifying components and on static relationships. Efforts on performance engineering are being made to represent software behavior and simulate it. However, no one has tried to extract a simulated model from existing software automatically.

We introduce SAAT, a tool developed at our research center. SAAT analyzes the dynamic aspects of software and creates a simulated model for performance analysis. We explain how the model can be generated, using a case study of UPnP middleware. This paper contributes to the bridge between performance analysis and reverse engineering

1. Introduction

Performance analysis is a process that analyzes dynamic execution flow, estimates the time and resources consumed, discovers potential bottleneck points, and predicts the performance in a real environment. In order to analyze software performance, information for such analysis should be provided by software architecture models and design specifications. This information is required to help understanding and predict time-dependent behaviors during performance analysis by dividing software into modules and by displaying time, intercommunication, data access frequencies, data transfer capacity of communication channels and other data.

If the existing software's design specifications are incomplete or incompatible with the current software version, the design specifications may not be used in the performance analysis. Additionally, development team members are sometimes too busy to participate in performance improvement work. In that scenario, a reverse engineering methodology will analyze the performance of the software. However, past studies in reverse engineering have concentrated on static aspects, which extract relationships among components through source code analysis. To date, fields of performance analysis and reverse engineering have not been directly related.

In this paper, we introduce the SAAT tool that will analyze and represent the dynamic structures of software visually for performance analysis. In Section 2, we introduce previous studies for software performance analysis and dynamic reverse engineering. In Section 3, we explain the basic concepts of SAAT. In Section 4, we explain the technological considerations to implement and the architecture of SAAT. In Section 5, we present a sample case of UPnP middleware. In Section 6, we discuss our results up to this point and any remaining problems and recommend tasks for future study.

2. Previous Studies

Researchers in performance engineering are studying how to integrate software architecture with performance information. In the realm of reverse engineering, dynamic reverse engineering to extract software execution models from existing systems is also being tried. In this section, we discuss the progress of research in these two areas, sharing the common factor of software modeling. We will survey research related to the software performance model (2.1) and we will cover the reverse engineering research status for existing system analysis (2.2).

2.1. Software Performance Model

The software performance model enables one to measure the detailed performance of software. In addition, the performance model allows quick and convenient structural investigation when problems are found. To allow this solution, the performance model shall precisely describe the system to be improved. Related researches including the following:

Smith [3,4] pointed out that there is no software architecture specification documented enough for performance analysis in general, and proposed the PASA (Performance Assessment of Software Architecture) methodology, which extracts architecture information from developer interviews and work products. PASA has 10 stages. In Stages 1 to 6, performance analysts examine software architecture and review the important use cases and scenarios with the development team. In Stages 7 to 10, the performance analysts construct and analyze the performance model, and announce the result. The PASA method requires dedicated cooperation from developers because the accuracy of the performance model depends on information provided by the developers.

Woodside [5,6] assumed that the contents that were not dealt with in software architecture documents, were omitted either because everyone understood the contents or they were something that didn't need to be described. He then presented the PASD (Performance Aware Software Development) methodology that produces and analyzes performance models from the design documents. The PASD has 7 stages. In Stages 1 to 3, performance-related information is added to the function-oriented specifications to make the specifications more complete. In Stages 4 to 5, the scenario model in the complete specification is transformed into a performance model. In Stages 6 to 7, performance is evaluated, and feedback is provided. In the PASD method, the performance model is created according to the specification's scenario model, and the accuracy of the specification affects the performance analysis.

Pooley [7] asserted that integration of performance factors with design methods shall precede the performance analysis framework and made efforts to integrate performance factors with UML notations. He also proposed simulation methods of the designmodels described in UML and performance analysis methods. The method presented by Pooley analyzes performance by producing simulation models with sequence diagrams, etc., used in dynamic modeling of UML or by changing using Petri-net models. Additionally, in Pooley's method, the accuracy of information given affects the performance analysis results of the model.

Similarly, researchers have made efforts to integrate the software performance model with software development methodologies and design models. If such efforts are connected with dynamic reverse engineering; more substantial effects can be achieved. First, it is possible to automate the creation of a performance model based on existing software. Accordingly, analysts might reduce time working with development team. In addition, the performance model does not need to rely on an incomplete design specification.

2.2. Software Reverse Engineering

In order to understand software, reverse engineering is used to identify software components and their interdependence and produces software design-level abstractions [8]. Software reverse engineering is being researched for various purposes, such as how to add new functions to existing software, maintain and improve system efficiency, and recycle modules in new systems. Recently, so-called dynamic reverse engineering has been started in an effort to discover software component interaction using software traces and records. The following discusses research related to the dynamic reverse engineering.

Systa[9,10] proposed the Shimba tool that automatically produces sequence diagrams of Java programs. With the Shimba tool, trace information is acquired while such programs are executed, and the information is then used to create a state diagram and a scenario diagram. Systa's papers give a lesson that dynamic aspects of software can be generated from monitoring software execution, but it does not propose to link the information to a performance model. Also, considering the fact that not many existing systems are constructed in Java, additional research is required for other languages.

Walker and Murphy [11,12] proposed an abstraction method, recognizing the fact that event trace information at the functional level presents a wide gap from the subsystem level of developers' interests. This method uses a visualization tool and a path query tool. The visualization tool shows a series of drawings according to system execution. The path query tool supports the analyze event flow information, using normal expressions that map the source codes to components of the developer's choice. The method presents the basic techniques in abstracting event trace information. However, the method seems to require some more time for field use, considering that it is limited to object-oriented languages, and no real application case has yet been presented.

Bengtsson and Bosch [13] pointed out that there was no research on architectural reengineering methodology, and if any, quality attributes were not considered. They defined a reengineering methodology based on scenarios. In this methodology, explicit and objective evaluation methods, such as simulations, mathematical model rings, etc., are adopted.

These efforts to produce architecture-level execution models from software execution flow have indeed begun [8,9,10,11,12,13]. The researches have presented many fundamental and useful results, but further efforts are required to make them practical such as expanding to the languages mainly used in real development. Also, there is still no attempt to connect the result to a simulation.

3. Concept of SAAT

The Software Architecture Analysis Tool was developed for performance analysis at the Software

Center, Samsung Electronics Co., Ltd. Since most of software programs are implemented in C language in Samsung electronics Co, Ltd., SAAT targets software constructed in C language. Our purpose was to overcome delays that accompanied performance analysis. Our activities of performance analysis are as follows: When a performance analyst is requested to analyze software performance, the analyst first has to understand the software's structure and its dynamic behaviors. Then, the analyst finds the component that unnecessarily consumes much time and resources. Finally, the analyst identifies improvement issues and solutions using a simulation tool. We hoped to shorten the time of the performance analysis in order to make the analyst's work more efficient. We tried to automate the analysis process.

The process of analyzing the software performance of the existing system at Samsung Electronics Co., Ltd., can be automated as shown in Figure 1. First, the information of how software modules interact should be recorded (Software Trace Data). Second, the interaction should be represented as nodes and edges in drawing a diagram (Behavior Model). If one would like to understand the dynamic structure of the software, a composite diagram explaining several interactions should be drawn (Execution Model). Last, the composite diagram should be converted for modeling in a simulation tool (Simulation Model). In the following subsections, we review the concepts of each model in Figure 1 in more detail.

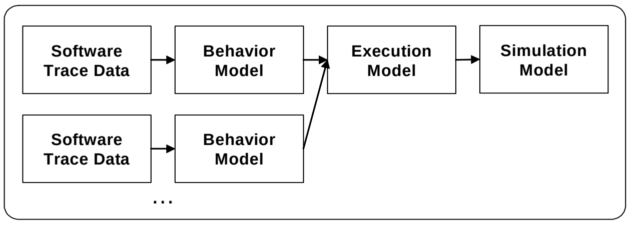


Figure 1: Automating process for performance analysis

3.1. Software Trace Data

Software Trace Data is a record of the interaction between the software modules. Because our target software is written in a procedural language, a software module is a function. Interaction involves not only call-relationships among several functions, but also information about the order of those functions. Hence, Software Trace Data includes function names, the calling relationship and execution order. Also, we add the function's running time into the data for performance analysis.

Software Trace Data is recorded as shown in Figure 2. The record consists of function name, time stamp, and flags according to the execution order. A line informing a function's start has information about function name, start time, and start flag. A line informing a function's end has information about function name, finish time, and finish flag. Also, lines of called functions are nested between the starting line and the finishing line of a calling function. As a result, both call-relationship and execution order can be represented in the Software Trace Data.

```
Function Main()
{
    call A();
    call B();
    call C();
}
Function A(parameter order)
{
    if (order == first_sequence)
    {
        call a();
        call b();
    }
    else
    {
        call c();
    }
}
Function B()
{}
Function C()
{}
```

```
A node  Time      Start/Finish

Main    start  time of Main  start  flag
  A     start  time of A     start  flag
    a   start  time of a     start  flag
    a   finish time of a     finish flag
    b   start  time of b     start  flag
    b   finish time of b     finish flag
  A     finish time of A     finish flag
  B     start  time of B     start  flag
  B     finish time of B     finish flag
  C     start  time of C     start  flag
  C     finish time of C     finish flag
Main    finish time of Main  finish flag
```

Figure 2: A concept of Software Trace Data

Software Trace Data is recorded while running the software. The easiest method for tracing software is to insert probing functions into functions of the target software. These probing functions write events of functions into a file when the functions start and finish. After compiling the software instrumented with probing functions, we can create a Software Trace Data file by running the software.

3.2. Behavior Model

The Behavior Model is a diagram representing Software Trace Data as nodes and edges. A node stands for a software module, namely, a function. Edges have two kinds of meaning. An edge directing below means the execution order between two connected functions. An edge directing to the right signifies a call-relationship between two linked functions. The Behavior Model is similar to a sequence diagram of UML, a design language for object-oriented programming. Thus, the Behavior Model represents one instance of the software executions.

Figure 3 shows an abstract presentation of the Behavior Model. Software modules are arranged according to their execution order and call relationship. In case "Main( )" calls "A( )," The Behavior model places "Main( )" left, "A( )" right, and shows an arrow starting from "Main( )" and arriving at "A( )." However, in another case, "Main( )" calls the second function, "B( )," Behavior model applies another rule: The second called function is connected to the first called function. Hence, the Behavior Model places "B( )" below "A( )" and shows an arrow starting from "A( )" to "B( )." The latter arrow stands for the execution order between “A( )” and “B( ).”

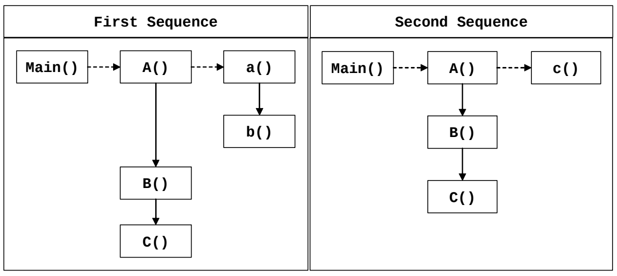


Figure 3: A concept of the Behavior Model

Rules of creating the Behavior Model from the Software Trace Data are related to start and finish flags. If a line has a start flag and the previous line also has a start flag, the node for the former is placed left of the node for the latter. Meanwhile, if a line has a start flag and the previous line has a finish flag, the node for the former is placed below the node for the latter. Additionally, if a line has a finish flag, there is no action for arrangement. The activity of arrangement occurs when reading a line including a start flag.

3.3. Execution Model

The Execution Model is a composite of several Behavior Models to represent the dynamic structure of the software. Even if each Behavior Model calls for the same function, the function may have different flows, based on control conditions and values of the variables. The Behavior Model can be combined by considering the control conditions, such as the branch or loop. To represent these control conditions, the Execution Model has several shapes of nodes. For example, the branch condition is represented as a rhomboid and the loop condition as a circle. We borrowed the concept of the Execution Model from the Execution Model of PASA[3].

The Execution Model can be extracted as shown in Figure 4 from the Behavior Models shown in Figure 3. A calling module nests called sub-modules. If a module has several different flows, the module possesses nodes showing the control conditions. For example, “A( )” has two different execution flows in Figure 3. Thus, “A( )” in Figure 4 has a branch node below the “Begin” node. After that, two different flows diverge from the branch node and converge to the “End” node in A( ).

The internal structure of functions may be extracted from former static reverse engineering tools. However, we have discovered that we can construct the Execution Model using only the information from the Behavior Models. While each Behavior Model is scanned, a branch node is inserted in case the Behavior Model has different part from the previous. When a function includes a repeated series of nodes, a loop node is added. Multiple Behavior Models are integrated to a single Execution Model in this way.

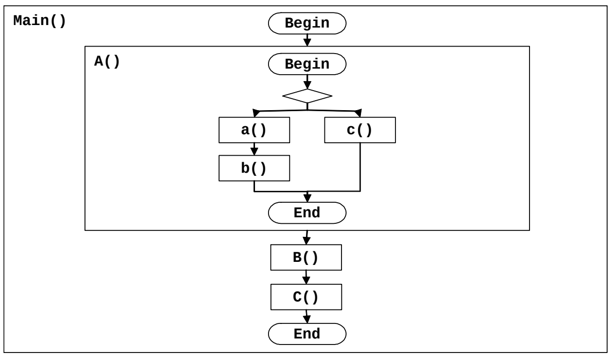


Figure 4: A concept of the Execution Model

3.4. Simulation Model

The Simulation Model is a model running in a simulation environment to demonstrate architectural issues of the present system. The Simulation Model is identical with the Execution Model in containing the information of software structure and performance information. Yet, we can modify, animate and simulate models in the simulation environment, so that we can predict a result of software implementation.

We can create the Simulation Model mapped from the Execution Model in figure 4 as follows. We delineate a model for “main ( )” in one module in our simulation tool. Next, we draw a model for “A( )” in a sub-model of the module separately. Finally, we create a transaction to trace models and insert a condition into the branch node in “A( )”. After that, we can simulate the models.

We can create the Simulation Model automatically by mapping each element of the Execution Model to each element of the Simulation Model. The Execution Model is composed of graphs, sub-graphs, nodes, and edges. The Simulation Model is composed of modules, sub-models, nodes, and arcs. Therefore, graphs are mapped to modules; sub-graphs are mapped to sub-models; nodes are mapped to nodes; and edges are mapped to arcs. The arrow in the Execution Model is mapped to a transaction of the Simulation Model.

4. Implementation of SAAT

In this section, we present the Software Architecture Analysis Tool (SAAT) – a tool to generate a dynamic model for the performance analysis of software. We developed the prototype of SAAT to examine the feasibility of SAAT projects.

The architecture of SAAT may be drawn as the Figure 5. SAAT is related to existing commercial tools. TAU provides trace data extracted from software execution [14], while aiSee is a tool showing a model in GDL [15]. aiSee allows users to see the Behavior Model and Execution Model. Finally Workbench conducts Simulation Modeling [16]. SAAT is also composed of three parts: One that extracts the Behavior Model from trace data (BM); one that integrates several behavior models into an execution model (EM); and one that changes the Execution Model to the Simulation Model (SM).

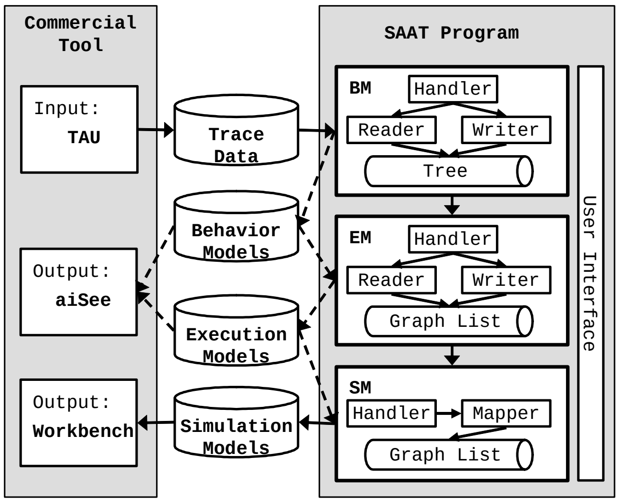


Figure 5: Structure of SAAT

4.1. Commercial Tools

4.1.1. TAU. The trace tool had to capture the system's response behavior to a user's request at regular intervals. Besides this basic role, we additionally had to consider multi-processors and multi-threads because current software consists of them. Fortunately, we found a reliable trace tool called TAU [14] for generating Software Trace Data. TAU supports multi-processors and multi-threads and generates data similar to the Software Trace Data in Figure 2.

4.1.2. aiSee. When we describe the Behavior Model and the Execution Model, we needed to determine a description tool. With respect to a description tool, we need a tool to present nodes and edges. Instead of implementing the tool, we searched for a convenient tool to present nodes and edges easily. We found that aiSee can present a diagram described in Graph Description Language (GDL). This language is a simple language to express graphs and has keywords like node, edge, graph, etc. [15].

4.1.3 Workbench. Since we use Workbench [16] as a simulation environment, we should convert the Execution Model to a model running in the Workbench. In the Workbench, one model is composed of nodes, arcs, and transactions. One model also could have several sub-models. By transforming the Execution Model into the simulation tool, we can obtain modeling data for simulating and analyzing our system's performance and verify its functionality.

4.2. SAAT Program

4.2.1 BM. When creating the Behavior Model, we should consider the abstraction level of the models so that users understand and manage the models easily. For that, the function lists should be categorized into modules to the level of the user's requirements. Also, low-level functions that users do not want to see should be eliminated or hidden to maintain the simplicity of the models. Thus, we endow a node possessing sub-nodes with a folding option to solve this issue. In spite of the importance of abstraction, the implementation was simple because aiSee supports the option with one token.

The drawing mechanism, used for the Behavior Model, is a binary tree. Each node in a binary tree has a left child and a right child. In the concept of Behavior Model, a left child becomes the first child, and a right childe turns to a sibling. The first child is the first called sub-function on the base of a function. A sibling implies the next executed function from a function. In reading, Reader saves information on the line to a node and pushes the node into a stack in temporary. When Reader reads a finish line related to the node in the stack, it pops the node and the previous node from the stack and links both to each other to make a tree. Then, Writer writes files by visiting from the root to leaf nodes in the tree.

4.2.2 EM. In creating the Execution Model, the internal structure of a function is needed because the Execution Model has a structure to fulfill the flows of several Behavior Models. For that purpose, if a function is called more than once and sub-operations of the function are different, loops and branches in the internal structure of the function need to be recognized. We replaced this structure of the function with the composition of the Behavior Models, as we wanted to get rid of cumbersome tasks like source analysis.

The structuring mechanism, used for the Execution Model, is a hash table that manages graphs that are organized in adjacency linked list. The hash key is a graph name. Once Reader reads tree information, Graph List recognizes nodes having a sub-tree, converts them to graphs, and saves the graphs in the hash table. After that process, Graph List scans the hash table. When Graph List finds the same name of a graph in the lists of graphs, it combines the two graphs into one. Finally, Writer creates a file of the Execution Model by scanning the hash table. These tasks occur several times if there are more Behavior Models.

4.2.3 SM. When creating the Simulation Model in Workbench from the Execution Model in GDL, we should deliberate on the conversion from one language to another. One parser may be required if we intend to convert GDL to Simulation language in Workbench. For that task, we can use Lex and Yacc for analyzing GDL and converting the language to Workbench graph codes. However, we omitted the subpart because we used a hidden file to have the concise information of the graphs instead of using the GDL file directly. Mapper reads information on the graphs in a hidden file of the Execution Model and maps the information to the Simulation Model. Writer outputs the files to run in a simulation environment.

5. Case Study

We applied SAAT to Universal Plug and Play (UPnP) [17], developed by Samsung Electronics Co., Ltd. UPnP is a home network middleware that supports distributed and open networks that are used to control devices and transmit data among devices. UPnP consists of two components: Controlled Device (CD) and Control Point (CP). CD provides services, while CP detects and controls the services. The middleware is implemented in C language.

Our concern was the feasibility of SAAT. We wondered whether the tool could be applied to ongoing development projects and products and how effective the result of creating a simulation model would be. For that, we captured the Trace Data of UPnP, using the TAU tool (5.1) and generated Behavior Models (5.2), Execution Model (5.3) and Simulation Model (5.4) in order.

5.1. Trace Data of UPnP

We captured the Software Trace Data of UPnP as Figure 6. We used TAU in order to instrument source files of UPnP with probing functions. The result files were five files containing the information of each thread because UPnP runs in five threads and TAU generates trace files according to each thread. However, we just show one trace file here as an example.

The case in Figure 6 is an instance of a concept in Figure 2, Section 3.1. TAU logs the execution time as 16 digits. With regards to flags, 1 means a start flag and -1 signifies a finish flag. This Software Trace Data has the information on the execution flow of UPnP. We can know which executions occurred in what order by looking at the list of functions by time. However, for a more intuitive understanding, such execution flow must be graphically represented.

```
[root@duri93 device]# cat events.0.edf
#  creation program: tau_convert -dump
#  creation date: jul-08-2003
#  number records: 40
#  number processors: 0
#  max processor num: 0
#  first timestamp: 1057631241510426
#  last timestamp: 1057631247598443

#=NO= =====EVENT== ==TIME [us]= =NODE= =THRD= ==PARAMETER=
   1                  (null)              1057631241510426    0    0
   2                  (null)              1057631241510447    0    0
   3          "int main(void) C  "        1057631241510456    0    0         1
   4 "void TvDeviceStateTableInit(v   1057631241510563    0    0         1
   5 "void TvDeviceStateTableInit(v   1057631241510590    0    0        -1
   6 "int UPnP_CD_Start(int, FunPtr   1057631241510696    0    0         1
   7         "int upnpStart() C  "        1057631241510732    0    0         1
                                  ............
  29 "void UPnP_CD_SetRenewTime1057631242718027    0    0        -1
  30 "int UPnP_CD_Finish(void) C  " 1057631244597264    0    0         1
  31      "void Stop_Threads() C  "     1057631244597326    0    0         1
  32 "int PrintString(char *, ...)      1057631244597332    0    0         1
  33 "int PrintString(char *, ...)      1057631244597366    0    0        -1
  34 "int PrintString(char *, ...)      1057631247598364    0    0         1
  35 "int PrintString(char *, ...)      1057631247598410    0    0        -1
  36      "void Stop_Threads() C  "     1057631247598418    0    0        -1
  37 "int UPnP_CD_Finish(void) C  "  1057631247598423    0    0        -1
  38          "int main(void) C  "       1057631247598428    0    0        -1
  39                  (null)              1057631247598440    0    0
  40                  (null)              1057631247598443    0    0
```

Figure 6: Tautrace.0.0.0.trc (Main flow of UPnP CD)

5.2. Behavior Model of UPnP

The Behavior Model connected to the Trace Data in Figure 6 is shown in Figure 7. This Behavior Model shows the execution order from the Trace Data of TAU. We can easily know that UPnP CD starts, sets cache control, sets CD's timeout, sets renew time, and finishes by reviewing the diagram in Figure 7. Each node can be folded or unfolded to hide or show a sub-tree of each node. Therefore, users can browse the Behavior Model at the level they want to know.

Behavior Models differ by the user services requested. Figure 8 shows another Behavior Model of UPnP. The cases in Figures 7 and 8 are those of the Behavior Model explained in Section 3.2. In addition to the concept of Section 3.2, information on the time consumed at each node is displayed beside the node name. Thus we can know the candidates for any bottleneck as well as the execution order and call-relationship.

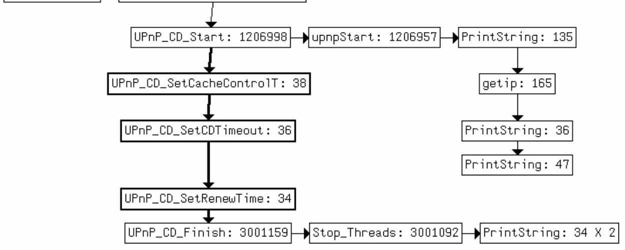


Figure 7: Behavior Model 1 (Main flow of UPnP CD)

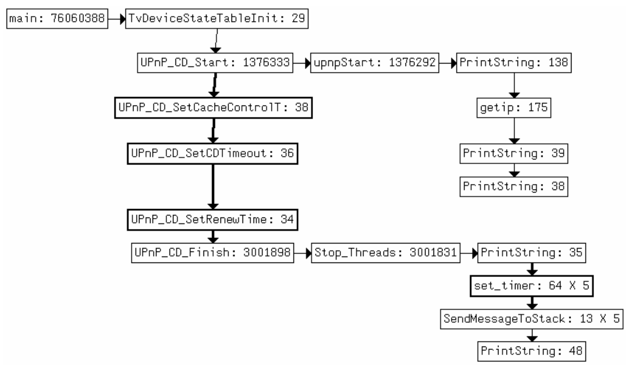


Figure 8: Behavior Model 2 (Main flow of UPnP CD)

5.3. Execution Model of UPnP

The Execution Model combines different Behavior Models. Figure 9 shows an Execution Model produced from the combination of Behavior Models of Figures 7 and 8 in Section 5.2. In Figure 9 below, the internal structure of “Stop_Threads( )” represents the Execution Model well. The “Stop_Threads ( )” presents a branch to two different internal flows and a loop showing repetition of the sub-flows.

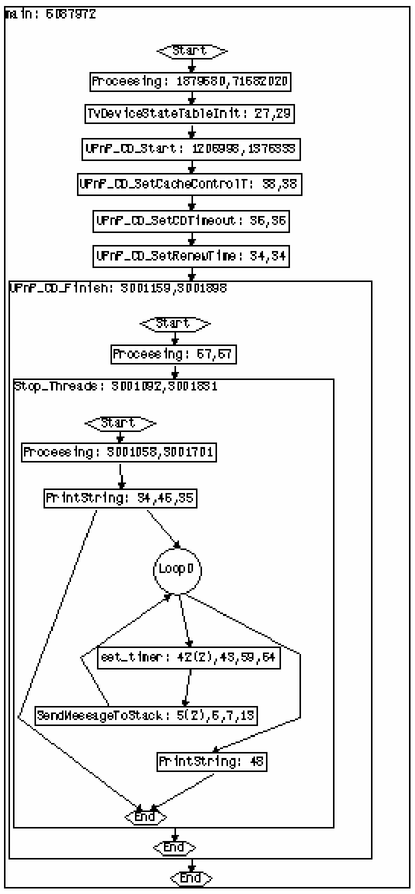

main: 5087972
Start
Processing: 1879580,71682020
TvDeviceStateTableInit: 27,29
UPnP_CD_Start: 1206998,1376333
UPnP_CD_SetCacheControlT: 38,38
UPnP_CD_SetCDTimeout: 36,36
UPnP_CD_SetRenewTime: 34,34
UPnP_CD_Finish: 3001159,3001898
Start
Processing: 57,57
Stop_Threads: 3001092,3001831
Start
Processing: 3001058,3001701
PrintString: 34,46,35
Loop0
set_timer: 42(2),43,59,64
SendMessageToStack: 5(2),6,7,13
PrintString: 48
End
End
End

Figure 9: Execution Model (Main flow of UPnP CD)

5.4. Simulation Model of UPnP

The Execution Model in Figure 9 is changed to the Simulation Model in Figure 10. The upper diagram in Figure 10 shows a Workbench model for the main function while the lower diagram shows a model for Stop_ Threads( ). The distinctive aspect in the Simulation Model is the transactions, which move dynamically as time goes. Therefore, we can modify and simulate a model to predict the performance of the system to be implemented and search for a better solution based on the simulation results.

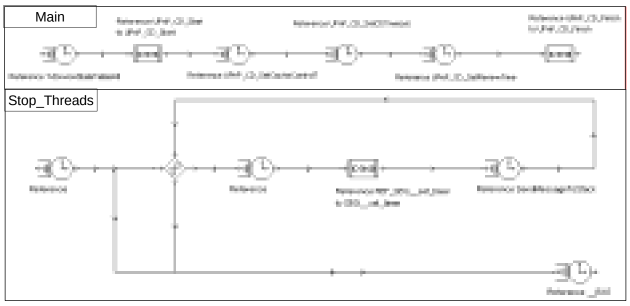


Figure 10: Simulation Model (Main flow of UPnP CD)

At this time, we confirm that the modeling data could be generated from monitoring software execution. However, the result of applying SAAT to the UPnP product is incomplete when comparing the result with that of a manual result for several reasons. First, SAAT pours all information into the Simulation Model, while performance analysts do not care for detailed layers. Second, the SAAT starts at the thread level while performance analysts start to model at the critical modules. Finally, SAAT does not distinguish concrete conditions, although some conditions could be important in modeling. We should develop SAAT to customize the Simulation Model according to user intention.

6. Conclusions

We have explained how we created the Execution Model from the execution trace data of the software system and how we constructed the Simulation Model from the Execution Model for performance analysis. Through SAAT prototyping, we showed that a simulation model could be automatically generated from the execution trace data of software. This case study shows the possibility of saving the time usually consumed in making a simulation model for performance analysis of software.

We propose the following additional research. First, we should find how to group functions that belong to the corresponding component. In this case, we may use the

Dali Workbench tool made by Kazman [18, 19]. However, we did not yet implement this method in SAAT. For this purpose, user intervention parts or important component-declaring parts must be added. In addition, we should complement SAAT by finding additional rules for converting from the Behavior Model to the Execution Model and by adding options to modify the Simulation model for user tastes. In the long term, we want to adapt this tool to several modeling environments.

7. References

[1] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, *Documenting Software Architecture: Views and Beyond*, Addison-Wesley, Sept. 2002.

[2] R.J. Pooley, "Software Engineering and Performance - a roadmap", *Proceedings of the conference on The future of Software engineering*, Limerick, pp189-200, July 2000.

[3] C.U. Smith and L.G. Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*, Addison-Wesley, Sept. 2001.

[4] C.U. Smith and L.G. Williams, "PASASM: An Architectural Approach to Fixing Software Problems", *Proc. CMG*, Reno, Dec., 2002.

[5] K. Siddiqui and M. Woodside, "Performance-Aware Software Development (PASD) Using Resource Demand Budgets", *Proc. of the 3rd WOSP*, Rome, July 2002.

[6] D. Petriu and M. Woodside, "Generating a Performance Model from a Design Specification", 3rd *Workshop on Generative Programming*, ECOOP 2001, June 2001.

[7] R. Pooley and P. King, "The Unified Modeling Language and Performance Engineering", *IEE Proceedings - Software*, Vol 146 No 1, pp 2-10, February 1999.

[8] E. Stroulia and T. Systä, "Dynamic Analysis For Reverse Engineering and Program Understanding", *Applied Computing Review*, ACM, vol 10, issue 1, 2002.

[9] T. Systä, "Understanding the Behavior of Java Programs", *Proc. of the 7th WCRE*, pp. 214-223, Brisbane, Australia, November 2000

[10] R. Kollmann, P. Selonen, E. Stroulia, T. Systä and A. Zündorf, "A Study on the Current State of the Art in Tool-Supporter UML-Based Static Reverse Engineering", *Proc. of the 7th WCRE*, pp.22-33, 2002.

[11] R. Walker, G. Murphy, J. Steinbok and M. Robillard, "Efficient Mapping of Software System Traces to Architecture Views", *CASCON*, 2000.

[12] R. Walker, G. Murphy, B. Free-Benson, D. Wright, D. Swanson and J. Isaak, "Visualizing Dynamic Software System Information through High-level Models", *Proc. of the 13th ACM SIGPLAN Conference on OOPSLA*, ACM Press, pp. 271-283, 1998.

[13] P. Bengtsson and J. Bosch, "Scenario-based Software Architecture Reengineering", *Proceedings of the 5th ICSR*, pp. 308-317, June 1998.

[14] University of Oregon, *TAU: Tuning and Analysis Utilities*, http://www.cs.uoregon.edu/research/paracomp/tau/, 1999.

[15] Absint, *aiSee*, http://www.aisee.com, 2002.

[16] *Workbench*, Hyperformmix, http://www.hyperformi-x.com/products/workbench.htm, 2003.

[17] Microsoft Corporation, *Universe Plug and Play Device Architecture*, http://www.upnp.org/download/UPnPDA10_20000613.htm, 2000.

[18] R. Kazman and S.J.Carriere, "Playing Detective: Reconstructing Software Architecture from Available Evidence", *CMU/SEI-97-TR-010*, Pittsburgh, PA: Software Engineering Institute, Canegie Mellon University, 1997.

[19] L. O'Brien and C. Stoermer, "Architecture Reconstruction Case Study", *CMU/SEI-2003-TN-008*, Pittsburgh, PA: Software Engineering Institute, Canegie Mellon University, 2003.

Performance Data Collection: Hybrid Approach

Edu Metz, Raimondas Lencevicius
Nokia Research Center
5 Wayside Road, Burlington, MA 01803, USA
Edu.Metz@nokia.com   Raimondas.Lencevicius@nokia.com

1. Introduction

As the complexity of embedded software systems grows, performance profiling becomes more and more important. Performance profiling of embedded software systems requires data collection with low overhead and high information completeness.

Performance profiling consists of monitoring a software system during execution and then analyzing the obtained data. There are two ways to collect profiling data: either event tracing through code instrumentation or statistical sampling. Event tracing may be more intrusive but allows the profiler to record all events of interest. Statistical sampling may be less intrusive to software system execution, but cannot provide complete execution information.

Our position is that data collection on embedded software systems should be performed using a hybrid approach that combines the completeness of event tracing with the low cost of statistical sampling. The following sections expand this position.

2. Performance Data Collection

Performance profiling determines where a software system spends its execution time. Performance profiling requires data collection during program execution. Such data collection can be done either by event tracing or by statistical sampling. Let us consider the implications of using these two methods.

2.1. Event tracing

Event tracing records events that occur during system execution. Event tracing can track various events, such as task switches, component entries and exits, function calls, branches, software execution states, message communication, input/output, and resource usage.

Tracing requires changes to the software system usually called *instrumentation*. Instrumentation can be inserted into various program representations: source code, object code, byte code, and executable code. Time wise, it can be inserted before program execution or during it. Adding trace instrumentation can be done manually, semi-automatically or automatically. Automatization of the instrumentation may be complex. Full discussion on complexities of automatic vs. manual instrumentation goes beyond the scope of this paper. It is sufficient to say that the instrumentation may be a burden-some task, especially if some manual work is needed.

Since an occurrence of any event creates a record, event tracing is characterized by the completeness of knowledge: if an event was recorded, it did occur; if it was not recorded, it did not occur. As we will see, this does not hold for statistical sampling. Performance engineers can also learn exactly when each event occurred since every record is time stamped. This allows a complete analysis of event relationships in time, for example, the measurement of precise time distance between any two events. A performance engineer using an event trace can reconstruct the dynamic behavior of a software system.

For example, consider energy consumption by a mobile device [4]. To map the software execution to the power consumed, a performance engineer needs to know exactly when a peripheral is started and stopped. The information from event tracing directly maps software execution and power consumption (Figure 1 shows the measured power consumption as a function of time and peripheral device activations/deactivations mapped onto the same timeline).

There are a number of difficulties in using event tracing. Users have to spend time instrumenting the software system. Event traces affect the performance of the software system distorting its execution [8].

Not only does event tracing take some time, adding traces changes the behavior of the software system because of additional memory accesses and input/output [6]. In real-time software systems, the instrumentation overhead can cause real-time constraint violations. Therefore, it is important to limit the intrusion by minimizing the instrumentation overhead [2][5]. One way to achieve this is by reducing the number of events traced. However, performance engineers have to choose carefully, since omitting events from tracing also reduces the amount of information available. For example, if only “on” and “off” events are traced in a peripheral, it is no longer possible to detect and map the peripheral’s different “on” modes to differences in the system’s power consumption. In choosing the instrumentation granularity it is important to address the trade-off between the amount of event information required and the performance impact of the trace instrumentation. This may be hard even for an experienced performance engineer.

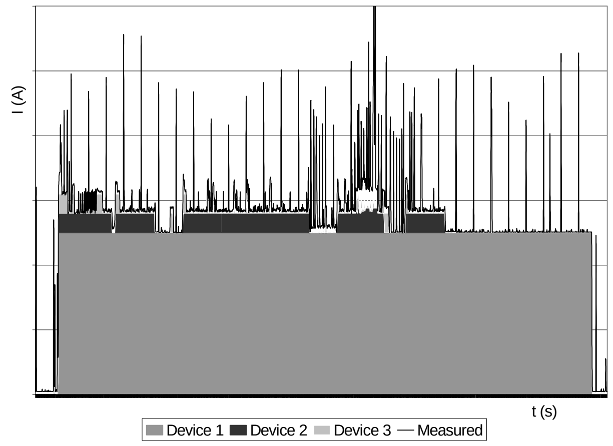


Figure 1. Device activations mapped to power consumption

For small routines, event tracing may not yield an accurate time comparison with larger routines. A small routine may suffer much higher relative overhead than a larger routine. If this is ignored, a great deal of effort may be wasted optimizing routines that are not real performance bottlenecks.

The data volume associated with event tracing can be very large: more than megabyte per second traced. This can cause a problem in devices that do not have large and fast storage or external network interfaces.

2.2. Statistical sampling

Statistical sampling relies on intermittent access to the software system to record its current state. Sampling can record different information: program counter (execution location), function call stack, scheduled or blocked tasks, active peripherals and so on. Sampling can be done strictly periodically or with certain randomness.

The simplest forms of sampling do not require any software modifications. A sampler simply copies the content of some processor registers to memory. In more complex sampling, the software system may need to be interrupted to record the needed information. In both of these cases, a performance engineer would usually spend much less time to achieve sampling than to instrument the software system for tracing.

The overhead of sampling may be orders below the overhead of tracing. For example, branch tracing may require overheads of over a factor of 10, function tracing may require overheads up to a factor of 2, while sampling at up to thousand samples a second may have an overhead of less than 1% [1]. (This estimation assumes a 100Mhz processor and 1000 cycles of work per sample, which is enough to read the address of the currently executed instruction and save this information. Using symbol information generated at compile time, the profiler can later correlate the recorded sample with the source code.) At such frequencies, sampling produces much less data than event tracing—a positive in storage-limited devices.

With advantages presented above, sampling is a perfect tool for gathering the performance data in systems where the low overhead is crucial. For example, sampling the execution of software in a mobile device executing real-time tasks may be the only way to obtain information about long-running functions without causing the software to miss real-time deadlines due to tracing overhead.

However, sampling also has downsides. The sampling frequency determines the granularity of the gathered information. In addition, the duration for which the software system executes directly relates to the number of samples collected. A sampling profiler requires software systems to execute over a reasonable period of time to ensure accuracy [7]. The goals of a performance engineer may require high sampling frequency that negates the low overhead and small data production of sampling.

Sampling yields only a statistical measure of the software's execution patterns. It does not provide completely precise numbers: if an event does not occur in a sampling log, there is no guarantee that it did not occur in execution. Therefore sampling may not be useful for situations that need to track exact numbers of events, for example, a singleton message to a task or an exact relationship between requests and acknowledgements. In periodic real-time systems, the sampling interval needs to be randomized to avoid sampling the same periodic software entity at every sampling point.

Sampling may not be able to detect frequently executed routines whose execution times are smaller than the sampling frequency. In addition, manual trace instrumentation usually tracks application-specific events that could be difficult to capture by sampling. For example, detecting a transition from a single-person voice call to a conference call may require event tracing.

Sampling is not a good approach when event causality is analyzed. Although it may extract a function call stack at the sample time, it cannot track all function calls or message exchanges. A performance engineer who needs a complete message sequence chart or component interaction graph might be better off choosing event tracing.

3. Hybrid Data Collection

Let us summarize the previous section. Event tracing yields the most detailed and complete system execution data. However, it takes time to instrument software, tracing has a high overhead and may change the behavior of the software system [6]. Statistical sampling is simple to use and less intrusive to software system execution, but does not provide causality relationships and exact data.

Embedded software systems, such as mobile devices, have real-time constraints and therefore require performance-profiling methods with low overheads. On the other hand, performance analysis of such devices often involves causality relationships and precision requirements. For example, a performance engineer needs to know exactly when a task starts processing a message in a multiplayer game that changes the game environment, since this may point to the cause of performance bottleneck evidenced by numerous file accesses.

Often neither event tracing nor statistical sampling can satisfy such conflicting requirements. The problem is further compounded by the fact that test runs are not entirely deterministic in mobile devices due to interactions with other systems such as mobile network elements. Therefore, performance data cannot be collected during multiple test runs, but instead needs to be collected during a single test run.

To collect performance data of embedded software systems with low overhead and adequate completeness, we propose to use a middleweight approach which is a hybrid of heavyweight event tracing and lightweight statistical sampling. Only a subset of all events is traced, providing limited completeness and causality information. Additional information is obtained through sampling.

To apply our method, a performance engineer has to determine which part of the performance data should be collected with event tracing and which with statistical sampling. The following subsections describe these choices using a couple of examples.

3.1. Processor time profiling

When the goal of a performance engineer is to determine which software components and subsystems spend most time running on a processor, statistical sampling can provide most information. It can reveal the approximate amount of time spent in a component, such as a task, module or function. Event tracing can supplement this information in a couple of areas. First, it can precisely identify switches of very high-level components, such as tasks. Second, it can demonstrate the component execution causality by tracking message exchanges. For example, consider the synchronization between tasks A and B in Figure 2. After sending message m1, task A enters a wait state where it waits for a state synchronization callback m2 from task B before continuing its execution. Here, event tracing can record and timestamp the sending of messages m1 and m2, while sampling can provide more in depth performance data during time intervals [t1, t2], [t2, t3], [t3, t4]. Just sampling is not enough to provide the crucial synchronization information.

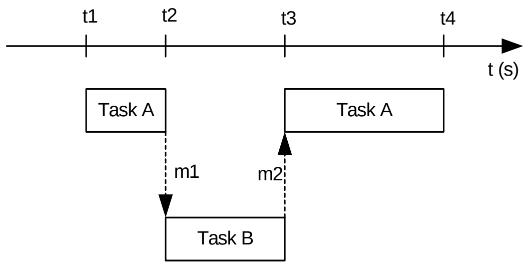


Figure 2: Task state synchronization

Profiling system interrupts requires event tracing as well. Even though the intrusion cost of tracing interrupts is high, sampling cannot be used here, because the execution times of interrupt handlers are much smaller than the sampling frequency.

3.2. Resource usage and energy profiling

In mobile devices power consumption varies depending on the peripherals used. During the system execution, software accesses peripherals. These accesses need to be recorded to determine when a peripheral is used. In resource usage and energy profiling, complete information about active and inactive peripherals is required. Event tracing needs to be used to track state transitions of Bluetooth, GPS or infrared subsystems. The intrusion cost of recording “on” and “off” events of peripherals is low since they occur infrequently.

Statistical sampling can complement event tracing by providing information that is too expensive to obtain using event tracing. For example, the processor power management puts the processor in a low power sleep mode when no software is scheduled to run. Unlike Bluetooth mode changes, the processor’s transition to the sleep state may be too frequent and too expensive to track via instrumentation. Statistical sampling can reveal the processor’s idle state with enough accuracy as long as the context switch time is an order of magnitude larger than the sampling frequency.

Another opportunity for sampling is presented by devices with multiple active modes. As mentioned in section 2.1, the overhead of tracing every state transition of a peripheral may be too high. While tracing could provide information about major “on” and “off” states, sampling could complement this information with infrequent samples of secondary states allowing more precise system mapping than achieved with just tracing.

3.3. Hybrid approach discussion

The proposed hybrid approach for performance data acquisition in embedded software systems has the potential to limit the data collection overhead while providing partial completeness and causality.

It is important to understand the requirements for performance data acquisition, which are domain and application specific. In different domains event tracing, statistical sampling, or our hybrid approach may provide the best solution. Our hybrid approach is sensitive to the choice of which performance data to collect using event tracing and which by statistical sampling. A couple of heuristics would be to trace infrequent events and non-deterministic events that provide causality information. However, further research is needed on how to make these choices.

The hybrid approach also yields the following benefits:

- Can provide useful profiling results in shorter execution runs than can be provided by pure statistical sampling.
- Can be used to profile events that occur infrequently.
- Limits the profiling data volume, which makes storing, transfer and post processing easier. Performance engineers are more likely to make use of profilers if they are easy to use.
- Allows reconstructing the dynamic behavior of a software system.

The proposed hybrid approach also has some limitations:

- Unless engineered intelligently, our hybrid approach could still inherit the drawbacks of both event tracing and statistical sampling.
- Trace instrumentation is still required, which may alter the behavior of the original software system.
- It yields two separate sets of profiling data. These two sources of information need to be combined and synchronized during post-mortem analysis.

Certain information could be reconstructed from statistical samples gathered during an execution. Events that deterministically precede events captured in a sample could be added to the performance data. This direction needs to be explored in future research.

4. Related Work

Several tools exist for performance profiling of software systems. Many of these are sampling based profilers [1]. Some tools, such as Intel's Vtune [9], provide event-tracing capabilities in addition to statistical sampling. However, the user cannot simultaneously use event tracing and statistical sampling during a single test run.

Hollingsworth et all [3] developed a hybrid data collection approach that uses event tracing to record state transitions in counter and timer data structures. These structures are then sampled periodically to collect performance data. Our hybrid approach uses event tracing to record a subset of all events of interest. The remainder of events is recorded through statistical sampling.

5. Conclusion

This paper describes a hybrid approach to the performance data collection. The hybrid approach involves striking a balance between event tracing and statistical sampling, combining the completeness of event tracing with low cost of statistical sampling. In addition, the proposed approach limits the profiling data volume. Useful profiling results can be obtained with relatively short execution runs.

We have described the use of a hybrid data collection approach for software execution time and resource consumption analyses. We believe that such an approach should be incorporated in future profilers. It is likely that other dynamic analysis domains would also benefit from incorporating both complete and sampling based data collection.

6. References

[1] J. Anderson , L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, R. Sites, M. Vandevoorde, C. Waldspurger, W. Weihl, Continuous Profiling: Where Have All the Cycles Gone?, *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 1997

[2] M. Arnold, B. Ryder, A Framework for Reducing the Cost of Instrumented Code, *Proceedings of the Conference on Programming Language Design and Implementation* (PLDI), 2001, pp. 168-179.

[3] J. Hollingsworth, B. Miller, J. Cargille, Dynamic Program Instrumentation for Scalable Performance Tools, *Proceedings of the Scalable High Performance Computing Conference,* 1994

[4] R. Lencevicius, E. Metz, A. Ran; Software Validation using Power Profiles, *Proceedings of the 20th IASTED International Conference on Applied Informatics* (AI 2002), Feb 2002.

[5] E. Metz, R. Lencevicius, Efficient Instrumentation for Performance Profiling, *Proceedings of the 1st Workshop on Dynamic Analysis,* 2003, pp. 143–148.

[6] D. Stewart, Measuring Execution Time and Real-Time Performance, *Embedded Systems Conference (ESC)*, 2001.

[7] K. Subramaniam, M. Thazhuthaveetil, Effectiveness of Sampling Based Software Profilers, *1st International Conference on Reliability and Quality Assurance*, 1994, pp. 1–5.

[8] J. Vetter, D. Reed, Managing Performance Analysis with Dynamic Statistical Projection Pursuit, *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, 1999.

[9] Vtune Performance Analyzer, March 2004. http://www.intel.com/software/products/vtune/

Using Runtime Information for Adapting Enterprise Java Beans Application Servers

Mircea Trofin[*]
Performance Engineering Laboratory
Dublin City University
mtrofin@acm.org

John Murphy[**]
Department of Computer Science
University College Dublin
j.murphy@ucd.ie

Abstract

Modern component-based technologies, such as Enterprise Java Beans (EJB), simplify system development by allowing developers focus on business logic, while system services are provided by an underlying application server. A class of system services, such as transactions or security, control the context in which components run.

The provisioning of such services can introduce performance overhead, as some system services might be executed redundantly. As EJB components bind dynamically, the determination that such an execution is redundant can be made only at runtime. We present a runtime mechanism for identifying and removing such redundant executions.

1. Introduction

Companies increasingly rely on component-oriented technologies, such as Enterprise Java Beans (EJB) [1], and Commercial Off-The-Shelf (COTS) components, in order to build large scale applications, reduce system development costs and capitalize on third party expertise.

Typically, component-based systems require an infrastructure that would support components, providing them with lifecycle services, intermediating component message interchange, etc. In the context of EJB, the infrastructure is referred to as an application server.

A trend in component-oriented technologies, especially the ones targeted at enterprise systems, is the separation of system-wide logic from business logic. That is, concerns such as security, transactional isolation, concurrency, or persistence (system concerns), are separated from concerns dealing with what actual services the application provides for its clients (business logic). This trend leads to a separation in responsibilities, as well: application server providers (vendors) are responsible for implementing system logic, leaving application developers with the responsibility of designing and implementing business logic. A module of such business logic is an EJB component, or bean.

Services such as transactions or security deal with the runtime context of a component, and they can be referred to as context management services. These services can be configured by means of deployment descriptors. Deployment descriptors are XML documents associated with each component, and include information indicating the configuration of context management services, on a per-method level. For example, for the security service, the configuration can indicate which user roles are allowed to execute the particular method.

Typical applications built on EJB include e-commerce and e-banking sites. Such applications are required to be highly available, while facing a potentially unbounded request rate. Another characteristic of these applications is that, while being multi-user, they tend to have little, if any, inter-user interaction, which makes the handling of various user requests highly parallel. In such cases, throughput is heavily impacted by the speed with which user requests are handled [2].

EJB applications are built by deploying EJB beans on an application server. The performance of such an application depends both on the characteristics of the developer-written code, as well as those of the application server. The code that ties components to the application server is called “glue code”; it acts as a proxy, calling application server services before and after calls to the component’s methods. In some cases, glue code is referred to as “container code”, however, since the concept of a container and the boundary between containers and application servers is not clearly separated in the EJB specification, we avoided using the term “container” in this paper.

Studies [3] have shown that a large proportion of the time spent to handle a client request is in fact spent within application server code. It is important, then, to optimize application servers in order to minimize their impact on performance.

Currently, the only means available for reducing the impact the application server has on performance is application refactoring [4]. Such refactorings can indeed improve performance, but at the cost of other system qualities, such as modularity or maintainability.

EJB components bind dynamically, at runtime. Based on how they bind, some context management services could be removed, in effect, minimizing the time spent within application server code. However, given the dynamic nature of EJB applications, the determination of

[*,**] The authors’ work is funded by Enterprise Ireland Informatics Research Initiative 2002.

what can be removed has to be done using runtime information. A more detailed presentation of this aspect of EJB has previously been made [5].

We present a solution for the analysis and removal of redundant executions of context management services between EJB components on an application server. The execution of a context management service is deemed redundant if the goal it tries to achieve has already been achieved by a previous execution. For example, if a transaction context is available, and the control is passed to a method requiring such a context, no additional effort is required for providing this context.

The effect of execution removals is the generation of new glue code versions for a component.

Our solution is generic with respect to context properties, i.e. the solution is not applicable only to the transactions and security services available in EJB.

Note that our effort is not concerned with dealing with remote method invocations in EJB, and all inter-component method invocations described here happen locally, within the same virtual machine. We are also concerned only with the cases where contexts are managed by the application server; EJB permits "bean-managed transactions" for example, which is a case we do not treat.

2. Solution Overview

The solution consists of extracting runtime information from an application and combining it with static information, to generate decisions as to which context management services are redundant.

To deem an execution redundant in a situation, it is necessary to know the context in which that execution is performed. That is influenced by the call path followed to this point, and the context requirements the previous methods in the path had. However, this is not sufficient at all times. Suppose that method m1 of component A indicates that only "admin" users can call it, and method m2 of component B indicates that only "manager" users can call it. Suppose m1 requests a binding to B and calls m2. Now suppose that happens under the credentials of user "Joe", who is an admin, as well as a manager. In this case, only one security context check is necessary, before m1 is called.

This would mean that we can only deem redundant service executions in cases when these services do not take into account the runtime value of the context. However, in the case of the example above, if it were known that "all admins are managers", the security context check at m2 could be deemed redundant. We call this kind of information "application specific facts".

A second problem that needs to be solved is accommodating the fact that the same component might participate in different binding scenarios in which different services might be deemed redundant.

We will first describe the structure of the information required for our solution, and then describe a system that uses this information to optimize an application server.

2.1. Representing Runtime Information: Binding Graphs

Binding graphs are a refinement over the runtime data the monitoring service is producing. A binding graph reflects the order in which bindings took place when a system client request was handled.

A binding graph is essentially a tree. Any node has at most one parent. One node is associated with a component. Each node has a list of method elements. Each method element has a list of binding elements. A binding element contains at most one node. This structure is depicted in Figure 1.

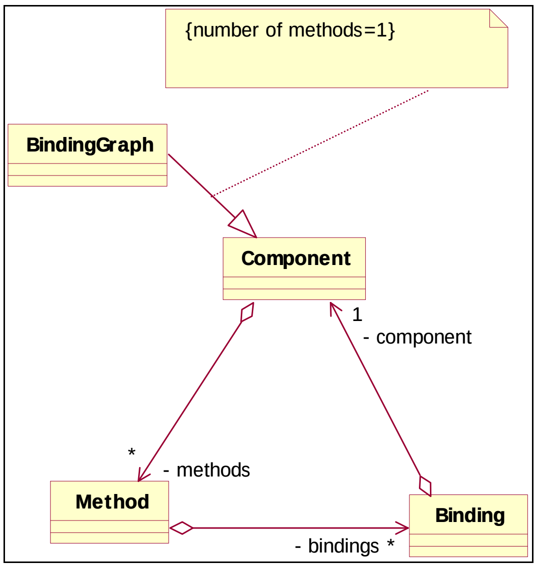


Figure 1 binding graph structure

An example of a binding graph is depicted in Figure 2. Nodes are illustrated as circles. An arc indicates the act of binding.

The root node has always only one element in the method set, as that is the method called by an external client. In our example (Figure 2), *method1* of component *A* eventually initiates a binding to *B* and then *C*. The order is not important, as both bindings happen in the context of *A*, and, since we assume that the context is not modified within a method, the contexts these bindings take place in are identical.

Next, *method1* of *B* initiates a binding to *D*. It is implied that *method1* of *B* was called by *method1* of *A*, since the binding arc ending in this node started there. Note how, because components *B* and *E* are being bound to twice, but in different contexts, therefore, they are being represented for each of those cases. For any node in the graph, the context it is bound in can be determined by identifying its parents, then tracing down the tree the binding process.

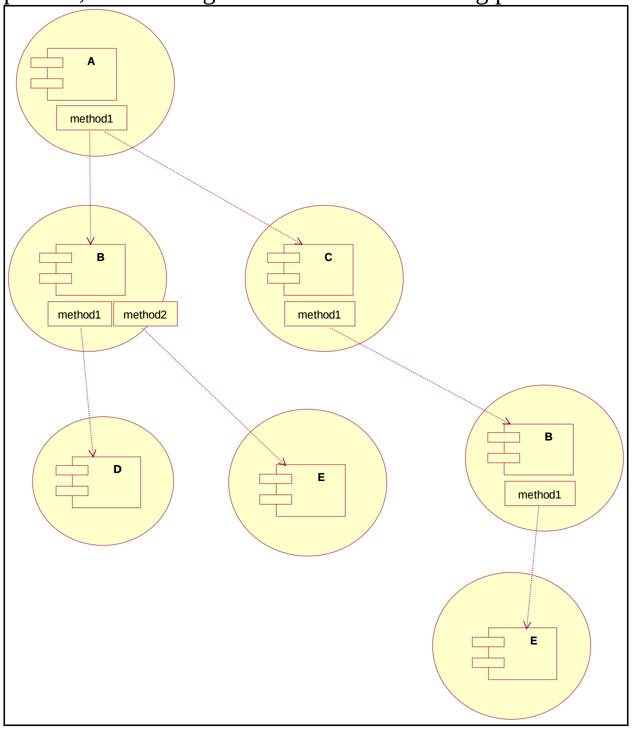


Figure 2 example of a binding graph

Note that a method element in a node describes the context in which calls to the methods in a child (bounded) node are performed. The purpose of the binding graph is to allow for the evaluation of the context in which components are used. For this reason, leaf nodes in a binding tree do not contain method elements, as they would not help evaluate anything (no further bindings). For the same reasons, if *method1* of component *A* called a method "*method0*" on component *B*, which did not lead to any further bindings, that information would not be represented in the tree.

In the case of a component calling its own methods, two possibilities exist: either the methods are called internally, without application server support (and no context management being performed), or through the application server, with context management. The latter would require a rebinding, which would appear in the binding graph as such. The former does not introduce any relevant information. Suppose *method1* of *C* calls *method2* of *C* internally, which in turn requests the binding to *B*. That binding still happens in the context of *method1*: since no application server support was used to call *method2*, no context management services are executed there.

2.1.1 Comparing Binding Graphs and Call Graphs

A call graph describes calls between various components in a system. Binding graphs filter out only those calls that lead to other components being bound. For this reason, more than one call graph can correspond to a single binding graph. Using our example in Figure 2, *method1* of component *A* can call some other methods on *B* after it binds to it, however, that is not important for our purposes, as they all happen within the context of *method1* of *A*. In fact, as it will be seen, what is optimized in this case is the complete glue code of *B*, given that any method *might* be called in the context of *method1* of component *A*.

2.2. Component Framework Rules

Deployment descriptors include information describing requirements placed on the context of execution by each method of a component. This information is encoded as configuration properties that affect the semantics of the execution of a corresponding context management service. For example, "transaction required" means that the method will be executed in the same transactional context as the caller, or, if that is not available, a new one will be created.

Currently, the set of possible configurations is published as part of the EJB specification and it's expressed in natural language. However, we can formally express them in a rule language, like Jess [6]. These rules describe how the context is transformed and whether something needs to be done to do that. For example, for "transaction required", the rule can indicate that, if no transaction context is available, the transactional context management service is to be run, and a new transactional context will be produced. We can refer to these rules as "component framework rules".

An example of such a rule is given in Figure 3.

```
(defrule transaction_required_noCtx
  (transaction required ?method)
  (not(transactionCtx))
  =>
  (assert (transactionSvc execute ?method))
  (assert (transactionCtx))
)
```

Figure 3 component framework rule example

The rule is written in Jess, a rule language similar in syntax to lisp. It describes what the "transaction required" configuration flag in any deployment descriptor means, in terms of executing the transaction context management service (transactionSvc) and in terms of the state of the context (transactionCtx). Here, the rule treats the case in which there is no transactional context available and so one has to be created. In order to achieve that, the transactions context management service has to be executed.

2.3. Context Requirements as Rule Engine Facts

Context framework rules determine a vocabulary that is used to describe the individual context requirements each method of a component. We will refer to such facts as context requirement facts. The translation between the syntax used for context requirements in deployment descriptors, and rule engine facts, is automatic. Translators can be reified using XSLT documents.

2.4. Application-Specific Facts

Relationships between security roles, as given in a previous example, constitute static information pertaining to a particular system. This information is encoded as facts, digestible by a rule engine. In our example, “all admins are managers” is such a fact. We will refer to these facts as application-specific facts.

2.5. Putting It All Together

The information in binding graphs, together with context requirement facts, describes a runtime scenario in terms of a succession of context requirements. Such information, together with application-specific facts, can be fed for processing by component framework rules in a rule-based engine. The output of the rules indicates which context management services need to be run. In other words, we have a mechanism for determining which services are redundant.

3. Solution implementation

Our focus is to develop a runtime optimization solution for application servers. It has to be easily integrated within existent application servers (R1). Extending it to support additional context management services should be done with minimal effort (R2). Implicitly, it is important to ensure that the overhead introduced by our solution does not exceed the performance improvements it generates (R3).

The optimization solution is able to analyze runtime information about an EJB application and decide in which cases context management service executions are redundant. This decision is based on both runtime information – binding grapsh, as well as static information pertaining to the system installation and the EJB framework – component framework rules, application-specific facts, and the information contained in deployment descriptors.

3.1. Overview

Our solution is implemented as an application server service, and consists of: a monitoring service that extracts runtime information from an application; a binding graph filter which extracts binding graphs from the runtime information produced by monitoring (Figure 4). An optimization coordinator controls the optimization of binding graphs by employing an expert system built on top of a forward chaining rule engine [7], such as Jess, which aggregates static and dynamic information and decides which services are redundant for a particular component. The glue code generator maps these decisions into the application server by generating specialized glue code variants. Finally, the call graph isolator ensures that glue code variants are called only in the situation they were optimized for.

Optimizations can be considered valid only for the period of time the set of components on an application server remains unchanged. Strategies for dealing with changes of the component set are under investigation; a trivial solution is to cancel all optimizations and start re-optimizing the system.

Our system is initialized with the set of component framework rules. Application-specific facts can be inserted, ideally pre-runtime, either manually, or automatically, if a facility is provided for that; however, this is outside the scope of our research.

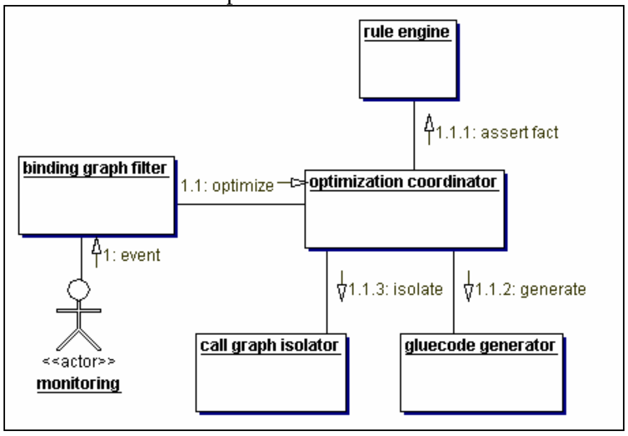


Figure 4 system overview

3.2. Overhead Considerations

Here we discuss aspects related to requirement R3. The optimization of a binding graph might be resource-consuming, but it occurs only once per graph. The overhead produced by our solution should be minimal, as most binding graphs should be optimized immediately after the application is started and serving requests. The more diverse the types of requests the server is presented with early-on, the faster the application will be fully optimized. Based on this observation, we can distinguish two different utilization scenarios of our solution (presented below). They differ in the period of time the optimizations take place. Since the active entity (the source of events) is the monitoring service, the differentiator between the two scenarios is the period of time the monitoring service is active.

3.2.1 Continuous Monitoring

In this scenario, monitoring is always active; therefore, optimizations can happen at any time. Since any new interaction is immediately optimized, the benefit is that all interactions end up optimized after the first time they are executed. The drawback is that monitoring imposes an overhead, which might not be desirable. This scenario is appropriate for the case in which the application under optimization is not well known, or in which monitoring is expected to be constantly turned on.

3.2.2 Training Period

In this case, monitoring is turned on for a period of time called training period, after which it's turned off. Therefore, optimizations can occur only during the training period. Ideally, the system would be exposed to as many different interactions as possible during this period, to minimize the number of un-optimized interactions left at the end of the training period. Insight into the system structure and behaviour is expected.

This scenario is appropriate for cases in which monitoring would not be normally turned on, and in which the application behaviour is well known. In such cases, it offers the benefit of having a fully optimized system (achievable during the training period), at no long-time extra performance cost due to monitoring.

3.3. Monitoring Service

The monitoring service extracts runtime events from an application, and makes them available to registered listeners. Such a listener is the Optimization Coordinator.

The development of this service is not part of our effort, as there are both academic [8] and commercial efforts in this area, which we can integrate with.

3.4. Binding Graph Filter

This component is tightly coupled to the monitoring service and processes whatever runtime information this service produces, extracting binding graphs. The tight coupling is due to the fact that there is no standard monitoring facility for EJB applications, and thus, the interface the various existing monitoring solutions offer needs to be adapted.

3.5. Optimization Coordinator

The optimization coordinator receives for processing one binding graph at a time from the binding graph filter. It maintains a set of binding graphs that it had optimized. Any binding graph is first checked against the optimized graph set. If it is not there (un-optimized), the binding graph is traversed depth first. It passes the context requirements of the method at the top to the rule engine, and then follows the first binding to the next node. Here, it passes all the context requirements of all the methods of the component associated with this node. At this stage, the rule engine decides, for each such method, which context management services are required.

The optimization coordinator invokes the glue code generator with these facts.

Next, the requirements of the methods are retracted, and we follow the next binding down by pushing the requirements of the method that owns the binding. The algorithm is presented in pseudocode in Figure 5.

```
Given RE, a rule engine

optimize (component c)
 for each method m in c

  push m's requirements in RE

  for each binding b in m
   c'= the component associated with b
   r=the set of requirements of the methods of c'
   push r in RE
   rc=get redundant context management executions
from rule engine, for c'
   generate glue code for c' given rc
   optimize(c')
   retract r from RE
  end

  retract m's requirements from RE

 end
end
```

Figure 5 optimization algorithm

Essentially, the algorithm generates a high-level specification of the glue code associated with a component, given a set of facts that can be known about the runtime environment that component might be run in.

3.6. Glue Code Generation

Requirement R1 governs the design of the integration between application server and the rule engine. There has to be minimal coupling between the rule engine and application server code, in particular, component glue code. However, we need to make some assumptions.

A strategy employed by some application servers, such as JOnAS [9], is to generate component glue code when the component is deployed. Usually, code templates are used, which are next run through a code generation engine, such as Velocity [10]. We developed our solution around the assumption that such a mechanism is being used.

The optimization coordinator has to use the information from the rule engine in order to generate specialized versions of glue code for each component. We opted to use a pre-processor solution. Within the code templates used to generate glue code, calls to context management services are tagged. Tagging can either be done with a technology such Velocity or XDoclet [11].

If the pre-processor is started with a set of properties, tagged areas of code can be excluded. Essentially, the code generation process is made aware of assumptions that can be made about the runtime environment of the code to be generated, which results in a customization of this code.

There has to be a mapping between facts produced by the rule engine and tags in the code. This can be ensured, as the "link" between these two is the set of component framework rules, which are available at the time glue code templates are developed.

3.7. Isolation of Call Graphs

In order to inject the optimized glue code back in the server, a major obstacle has to be overcome: the fact that the same component can participate in different interactions, which in turn can yield different glue code optimizations.

Our solution is to provide variants of glue code simultaneously, for the same component, and provide client components with a selection mechanism that allows them to pick the correct variant. A glue code variant of a component *A* is "correct" with respect to a client *C* in the following sense: consider the binding graph *B* that, through optimization, leads to the production of the glue code variant *GcV-A*, for component *A*. Let *B'* be the binding graph associated with the call path in which *C* is part of. If *C* has the same position (same parents) in *B'* and *B*, and *C* tries to bind to *A*, then *GcV-A* is the correct variant.

Refer to Figure 2. In that case, all external clients would bind to *A* via a variant of glue code dedicated to such clients. In this particular case, when *A*'s *method1* is called, the glue code installs a specialized naming provider. When an attempt is made to bind to *B*, this naming provider returns a reference to *B*'s glue code variant which optimizes for the current situation (i.e. *A*'s *method1* binding to *B*). This glue code variant of *B* installs a naming provider when *B*'s *method1* is called which "knows" which version of *D*'s glue code to chose; similarly, for *method2* and component *E*.

The call graph isolator requires the modification of the application server in order to allow for multiple glue code variants.

3.8. Extensibility

To extend our solution to support other context management services (R2), the component framework rule set has to be updated, and glue code templates need to be tagged accordingly. The ability of extending our solution is not so much targeted at EJB applications, as more to the migration of our solution to other frameworks, similar to EJB, such as CCM [12].

4. Related Work

4.1. Operating Systems

Context switching optimizations were analysed in the domain of operating systems (OS). For example, the authors of [13] optimize thread-related context switching overhead, by analysing liveliness information of context elements (such as registers). In [14], the authors attempt to avoid context switching incurred at inter-process communication.

There are two core differences between context switching optimizations in the OS area and our effort, which spawn from differences in problem domains. One lies with the entity that controls the context. In the OS case, the context of execution of a process is represented by a set of values (registers, stack pointer, etc) that belong to the process in the sense that it is the one that alters/controls them. The OS only saves and loads such values, but does not control them. In the components case, contexts are completely out of the scope of a component's control. The context is constrained outside the component's code, and is managed by the platform (application server). This allows for greater opportunities for analysis and optimization in the components case, as all the information related to context management can be made accessible by the platform to the agent performing the optimization.

The other difference lies with the composition of the context being managed. In the case of operating systems, this composition is "a given"; it typically consists of CPU registers. In general case we are focusing on, the composition of the contexts is variable.

4.2. Programming Language Compilers

The area of code optimization, including redundancy elimination, in the context of compilers, has been under extensive research and has achieved maturity. Currently, the vast majority of programming languages are compiled by compilers that make use of optimizers. In the case of interpreted languages, or languages that run over virtual machines, as the case is for Java, the virtual machine can provide an additional set of optimizations for a program.

Optimizations operate on information that is extracted from code, and, sometimes, on information related to the target platform. Typically, an intermediate representation is produced, on which optimization algorithms are run. The result is a modified representation, which is functionally equivalent to the first one, but optimizes for a particular aspect (i.e. space, time)

One requirement for redundancy elimination algorithms is that full data flow information be available [15]. In the case of EJB applications, this is generally impossible, as the execution can be distributed: for example, security checks could happen on a remote machine.

4.3. Optimization of Component Systems

The authors of [16] propose to optimize a component system at runtime. Their approach consists of recompiling an application built out of components, as interactions between components become apparent. The system is continuously evaluated and recompiled. Initial results indicated that a continuous evaluation-recompilation cycle is performance-detrimental.

The authors of [17] suggest that specialization scenarios for components be packaged together with components. The methods of specialization suggested are at the code level.

The most important difference between these approaches and ours is that code-level optimizations will miss out the semantics of context management services. We believe that our approach and the ones presented here can be applied conjointly, but they will optimize different aspects of the application in cause.

A number of authors propose that application servers offer facilities that would allow applications adapt to changes in their environment. An example is the work presented in [18]. Enterprise services tied to an EJB application server can be added/removed or altered. This is similar to what we propose, in a sense, as the effect of our optimizations is that the set of services that gets executed at inter-component calls gets altered. The difference lies with the scope of the alteration: in our case, it is specific to a particular interaction scenario in which a particular component participates, and is done in response to the discovery of redundant context management service executions; in [18], modifications affect all such interactions, and are performed as response to a change in the application environment (such as battery power or network conditions).

JBoss [19] offers the capability of adding or removing services provided by the application server for a component. Similar to the approach above, this capability has the shortcoming of affecting all interactions with that component. This approach cannot support the case in which the same component participates in different execution contexts.

5. Current Status and Future Work

5.1. Optimization Study

We conducted feasibility tests for our rule engine based optimization solution. We started by defining component framework rules for the transaction service, and extracting context requirements as facts from a set of components. No application-specific facts were used at this time.

We chose the transactions service for this test as it offers a larger array of configuration options, when compared to security.

The experience supports the current solution. The next immediate step is to include security rules, together with application-specific facts.

5.2. Call Graph Isolator Implementation

We have implemented a prototype call graph isolator on JBoss. We have used this prototype in order to gain more insight into the design implications related to it, as well as verify whether such a mechanism would introduce any overhead. A full discussion of the isolator is out of the scope of this paper. JBoss was used for this prototype purely for previous experience reasons. Since JBoss uses a reflective approach to glue code, it is not suited for implementing the rest of our solution; however, it proved sufficient for the purpose of this prototype.

5.3. Future Work

We intend to finalize a prototype optimization coordinator and engine, together with the corresponding set of rules for transactions and security services, as supported by EJB.

The next step will be to analyse the proposed glue code generation mechanism, in terms of technology used. Its applicability across various application servers will also be analysed. As we assume a particular glue code generation style in-place (template-based code generation), we will analyse glue code generation solutions for other cases - JBoss, for example, employs a reflective approach.

6. Conclusions

We presented the problem of determining which context management service executions are redundant for applications built on the Enterprise Java Beans component framework.

The proposed solution consists of aggregating static and dynamic information and producing variants of glue code that contain only context management service calls that are not redundant. Static information consists of component framework rules, context requirement facts, and application-specific facts. Dynamic information is encoded in binding graphs. The decision as to which service calls are redundant is made by a rule-based engine.

Glue code variants are produced by augmenting a currently employed method, template-based code generation.

Glue code variants are bound to the situation they are specialized for (i.e. a particular position in a call graph). A method has been presented and prototyped for ensuring that this binding is respected every time calls are passed between components.

7. References

[1] Sun Microsystems. “Enterprise Java Beans Specification”, http://java.sun.com/products/ejb/docs.html#specs

[2] The Middleware Company Case Study Team. “J2EE and .Net (Reloaded). Yet Another Performance Case Study”. http://www.middleware-company.com/casestudy/tmc-performance-study-jul-2003.pdf . June 2003

[3] E. Cecchet, J. Marguerite, W. Zwaenepoel. “Performance and scalability of EJB applications”. In Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) November 2002, Seattle, WA

[4] Brett McLaughlin. “Building Java Enterprise Applications Volume I: Architecture”. O’Reilly, 2002

[5] Mircea Trofin, John Murphy. “A Self-Optimizing Container Design for Enterprise Java Beans Applications”. The 8th International Workshop on Component Oriented Programming (WCOP), part of the 17th European Conference on Object-Oriented Programming (ECOOP). July 2003, Darmstadt, Germany.

[6] Sandia National Laboratories. Jess, the Rule Engine for the Java™ Platform.

[7] S. Russell, P. Norvig. “Artificial Intelligence. A Modern Approach”. Prentice-Hall, 1995

[8] Adrian Mos, John Murphy. “Performance Management in Component-Oriented Systems using a Model Driven Architecture Approach”. In proceedings of The 6th IEEE International Enterprise Distributed Object Computing Conference (EDOC), September 2002, Lausanne, Switzerland

[9] ObjectWeb. “JOnAS: Java™ Open Application Server”. http://jonas.objectweb.org/

[10] The Apache Jakarta Project, “Velocity”. http://jakarta.apache.org/velocity/

[11] XDoclet – Attribute Oriented Programming. http://xdoclet.sourceforge.net/

[12] Object Management Group. “Corba Component Model” http://www.omg.org/technology/documents/formal/components.htm

[13] Dirk Grunwald, Richard Neves. “Whole-program optimization for time and space efficient threads”. In Proceedings of the seventh international conference on Architectural support for programming languages and operating systems, Cambridge, Massachusetts. 1996

[14] Erik Johansson, Sven-Olof Nystrom. “Profile-guided optimization across process boundaries”. In Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization, 2000

[15] Steven S. Muchnick. “Advanced Compiler Design Implementation”. Morgan Kaufmann Publishers, 1997

[16] A. Gal, P.H. Fröhlich, M. Franz. “An Efficient Execution Model for Dynamically Reconfigurable Component Software”. In Seventh International Workshop on Component-Oriented Programming (WCOP 2002) of the 16th European Conference on Object-Oriented Programming. June 2002, Malaga, Spain

[17] Gustavo Bobeff, Jaques Noye. “Molding Components Using Program Specialization Techniques”. In Eight International Workshop on Component-Oriented Programming (WCOP 2003) of the 17th European Conference on Object-Oriented Programming. July 2003, Darmstadt, Germany

[18] Zahi Jarir, Pierre-Charles David, Thomas Ledoux. “Dynamic Adaptability of Services in Enterprise JavaBeans Architecture”. In Seventh International Workshop on Component-Oriented Programming (WCOP 2002) of the 16th European Conference on Object-Oriented Programming. June 2002, Malaga, Spain

[19] The JBoss Group, “JBoss Administration and Development Documentation – eBook - 3.2.1”. http://www.jboss.org

Efficient Specification-Assisted Error Localization

Brian Demsky Cristian Cadar Daniel Roy Martin Rinard
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139
{ bdemsky, cristic, droy, rinard }@mit.edu

ABSTRACT

We present a new error localization tool, Archie, that accepts a specification of key data structure consistency constraints, then generates an algorithm that checks if the data structures satisfy the constraints. We also present a set of specification analyses and optimizations that (for our benchmark software system) significantly improve the performance of the generated checking algorithm, enabling Archie to efficiently support interactive debugging.

We evaluate Archie's effectiveness by observing the actions of two developer populations (one using Archie, the other using standard error localization techniques) as they attempted to localize and correct three data structure corruption errors in a benchmark software system. With Archie, the developers were able to localize each error in less than 10 minutes and correct each error in (usually much) less than 20 minutes. Without Archie, the developers were, with one exception, unable to locate each error after more than an hour of effort.

1. INTRODUCTION

Error localization is a key prerequisite for eliminating programming errors in software systems and, in many cases, the primary obstacle to correcting the error — the fix is often obvious once the developer locates the code responsible for the error.

The primary issue in error localization is minimizing the time between the error and its manifestation as observably incorrect behavior. The greater this time, the longer the program executes in an incorrect state and the harder it can become to trace the manifestation back to the original error. This issue can become especially problematic for data structure corruption errors — these errors often propagate from the original corrupted data structure to manifest themselves in distant code that manipulates other derived data structures, obscuring the original source of the error.

This paper presents a new error localization tool, Archie[1], describes the optimizations required to make Archie efficient enough for practical use, and discusses the results of a case study we performed to evaluate its effectiveness in helping developers to localize and correct data structure corruption errors. Our results indicate that, after optimization, Archie executes efficiently enough for interactive use on our benchmark software system and that it can dramatically improve the ability of developers to localize and correct injected data structure corruption errors in this system.

1.1 Consistency Checking

Consistency checking is currently used as a technique for debugging [17]. Developers sometimes hand-code consistency checks in the same programming language as the rest of the system. The complication is that developers must code data structure traversals and implement any auxiliary data structures required to check the desired properties. Developing this code can be especially difficult because the developer cannot assume that the data structures satisfy any property at all — the whole point of the checker is to detect data structures that may arbitrarily violate their invariants. For example, straightforward hand-coded tree traversals may fail to terminate on trees that contain cycles.

Hand-coded consistency checkers are also vulnerable to anomalies such as incomplete property coverage, unwarranted assumptions about the input data structures, and increased development overhead. Our experience indicates that hand-coded consistency checkers are substantially larger and more difficult to develop than an equivalent consistency specification.

1.2 Specification-Based Approach

Archie accepts a specification of key data structure consistency properties (including sophisticated properties characteristic of complex linked data structures), then periodically monitors the data structures to detect and flag violations of these properties. The developer (potentially assisted by an automated tool) places calls to Archie into the software system. If the system contains a data structure corruption error, Archie localizes the error to the region of the execution between the first call that detects an inconsistency and the immediately preceding call (which found the data structures to be consistent).

Each Archie specification contains a set of model definition rules and a set of consistency properties. Archie (conceptually) interprets these rules to build an abstract model of the concrete data structures, then examines the model to find any violations of the consistency properties. The conceptual separation of the specification into the model construction rules and consistency constraints simplifies the expression of the consistency constraints and provides important expressibility benefits. Specifically, it enables the specification developer to 1) classify objects into different sets and apply different consistency constraints to objects in different sets, 2) express the consistency constraints at the level of the concepts in the domain rather than at the level of the (potentially heavily encoded) realization of these concepts in the concrete data structures, 3) use inverse relations to express constraints on the objects that may refer (either directly or conceptually) to a given object, 4) construct auxiliary relations that allow the developer to express constraints between objects that are separated by many references in the data structures, and 5) express constraints involving abstract relationships such as object ownership.

1.3 Optimizations

It is clearly desirable to perform the consistency checks as frequently as possible to minimize the size of the region of the execution that may contain the error. The primary obstacle is the check execution overhead. We found that our initial implementation of the consistency checking algorithm as described above was too inefficient for practical use. We therefore implemented the following optimizations:

- **Fixed-Point Elimination:** The Archie compiler analyzes the dependences in the specification to, when possible, replace the fixed-point computation in the model construction phase with a more efficient single-pass algorithm.
- **Relation Elimination:** The compiler analyzes the specification to, when possible, replace the explicit construction of each re-

[1]Archie is named after Archie Goodwin, the assistant to Rex Stout's fictional detective Nero Wolfe. The idea is that, under Wolfe's direction, Archie does all the work required to localize the crime to a specific suspect, then Wolfe uses his superior intelligence to solve the crime.

lation with a computation that efficiently generates, on the demand, the required tuples in the relation.

- **Set Elimination:** The compiler analyzes the specification to, when possible, integrate the consistency checking computation for each set of abstract objects into the data structure traversal that (in the absence of optimization) constructs that set. The success of this optimization enables Archie to eliminate the construction of that set.

Together, these optimizations make Archie run over 800 times faster on our benchmark software system than the original compiled version; the fully optimized instrumented version executes less than 6.2 times slower than the original uninstrumented version. For our benchmark software system, the optimized version of Archie is efficient enough to be used routinely during development with more than acceptable performance for interactive debugging.

1.4 Case Study

To evaluate Archie's effectiveness in supporting error localization and correction, we obtained a benchmark software system, used manual fault injection to create three incorrect versions, then asked six developers to localize and correct the errors. Three developers used Archie; the other three used standard techniques.

With Archie, the developers were able to localize each error within several minutes and correct the error in (usually much) less than twenty minutes. Without Archie, the developers were (with a single exception) unable to localize each error after more than an hour of debugging. The key problem was that continued execution made the errors manifest themselves far (in both code and data) from the original source of the error. Although the developers eventually came to understand what was going wrong, they were unable to trace the manifestation back to its root cause within the allotted time.

To place these results in context, consider that our benchmark system contains significant numbers of assertions designed to catch data structure corruption errors, two of the three errors manifest themselves as assertion violations, but these assertions were still not enough to enable the developers to locate the errors in a timely manner. These results indicate that Archie can provide a substantial improvement over standard error localization techniques.

1.5 Contributions

This paper makes the following contributions:

- **Archie:** It presents the design, implementation, and evaluation of Archie, a new specification-based data structure consistency checking tool for error localization and correction.
- **Optimizations:** It presents a set of optimizations (fixed point elimination, relation elimination, and set elimination) that, together, increase the performance of Archie on our benchmark software system by over a factor of 800, enabling Archie to be used routinely during interactive development with more than acceptable performance.
- **Case Study:** It presents a case study that evaluates the effectiveness of Archie as an error localization and correctness tool. With Archie, developers were able to quickly localize and correct errors in our benchmark software system; without Archie, developers were unable to localize the errors even after they spent significant amounts of time attempting to trace the manifestation of the errors back to their root causes.

2. EXAMPLE

We next present an example (inspired by the FreeCiv program discussed in Section 6) that illustrates how Archie works. The program maintains a grid of tiles that implements the map of a multiple-player game. Each tile has a terrain value (i.e. ocean, river, mountain, grassland, etc) and an optional reference to a city that may be built on that tile. Figure 1 presents the relevant data structure definitions.

```
structure city { int population; }
structure tile { int terrain; city *city; }
tile grid[EDGE * EDGE];
```

Figure 1: Structure Definitions

```
set TILE of tile
set CITY of city
relation CITYMAP: TILE -> CITY
relation TERRAIN: TILE -> int
```

Figure 2: Set and Relation Declarations

```
for x=0 to EDGE*EDGE, true => grid[x] in TILE
for t in TILE, true => <t,t.terrain> in TERRAIN
for t in TILE, !t.city = NULL =>
    <t,t.city> in CITYMAP
for t in TILE, !t.city = NULL => t.city in CITY
```

Figure 3: Model Definition Rules

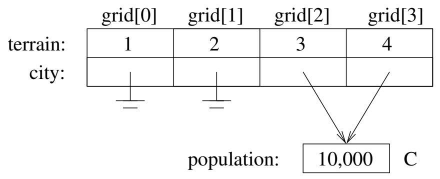


Figure 4: Concrete Data Structure

TILE = $\{grid[0], grid[1], grid[2], grid[3]\}$
TERRAIN = $\{\langle grid[0], 1\rangle, \langle grid[1], 2\rangle, \langle grid[2], 3\rangle, \langle grid[3], 4\rangle\}$
CITY=$\{C\}$
CITYMAP=$\{\langle grid[2], C\rangle, \langle grid[3], C\rangle\}$

Figure 5: Model Constructed for Example

Even a data structure this simple has important consistency constraints; in this section we focus on the following constraints: the terrain field of each tile contains a legal value, each city is referenced by exactly one tile, and no city is placed on an ocean tile.

2.1 Expressing Consistency Properties

To express these constraints, the developer first identifies the sets and relations that conceptually model the concrete data structures. In our example there are two sets, `TILE` and `CITY`, and two relations, `CITYMAP` and `TERRAIN`. Figure 2 presents the declarations of these sets and relations. The `TILE` set contains `tile` structures, and the `CITY` set contains `city` structures. Each relation consists of a set of tuples with objects from two specified sets.

2.1.1 *Model Definition Rules*

The developer next provides a set of model definition rules that define a translation from the concrete data structures to the sets and relations in the model. Figure 3 presents the model definition rules in our example. Each rule consists of a quantifier that identifies the scope of the rule, a guard that must be true for the rule to apply, and an inclusion constraint that specifies an object (or tuple) that must be in a given set (or relation). Conceptually, Archie uses a least fixed-point algorithm to repeatedly add objects to sets and tuples to relations until the model satisfies all of the constraints. For the data structure in Figure 4, Archie constructs the model in Figure 5.

2.1.2 *Consistency Constraints*

The developer next uses the sets and relations to state the consistency constraints. Each constraint consists of a sequence of quantifiers that identify the scope of the constraint and a predicate that the constraint must satisfy.

Figure 6 presents the constraints in our example. The first constraint ensures that each tile has a valid terrain, the second ensures that each city has exactly one location (i.e., exactly one tile references each city), and the final constraint ensures that no city is placed on

```
for t in TILE, MIN <= t.TERRAIN and t.TERRAIN <= MAX
for c in CITY,sizeof(CITYMAP.c)=1
for c in CITY,!(CITYMAP.c).TERRAIN = OCEAN
```

Figure 6: Consistency Constraints

an ocean tile. [2] As this example illustrates, the ability to freely use inverses substantially increases the expressive power of the specification language — it enables the expression of properties that navigate backwards through the referencing relationships in the data structures to capture properties that involve both an object and the objects that reference it.

2.2 Instrumentation and Use

Finally, the developer (potentially with the aid of an automated tool) instruments the code to periodically invoke Archie, which examines the data structures and reports any inconsistencies to the developer. When the instrumented program executes, Archie localizes the error to the region of the execution between two subsequent calls to Archie and identifies the violated constraint (which, in turn, identifies the corrupt data structure). Our results (as discussed in Section 6) show that this approach can enable the developer to quickly localize and correct the error that caused the inconsistency. With standard approaches, the program typically continues its execution for some period of time, with the error propagating through the data structures. This combination of continued execution and error propagation makes it difficult to understand and localize the error.

3. SPECIFICATION LANGUAGE

Our specification language consists of several sublanguages: a structure definition language, a model definition language, and a model constraint language.

3.1 Structure Definition Language

The structure definition language supports the precise specification of heavily encoded data structures. It allows the developer to declare structure fields that are 8, 16, and 32 bit integers; structures; pointers to structures; arrays of integers, packed booleans, structures, and pointers to structures. The array bounds can be either constants or expressions over an application's variables. The developer can declare that a region of memory in a structure is reserved, indicating that it is unused. Finally, the structure definition language supports a form of structure inheritance. A substructure must have the same size and contain all of the same fields as the superstructure, but it may define new fields in areas that are unused in the superstructure.

3.2 Model Definition Language

The model definition language allows the developer to declare the sets and relations in the model and to specify the rules that define the model. A set declaration of the form `set S of T: partition` $\mathtt{S}_1, ..., \mathtt{S}_\mathtt{n}$ declares a set `S` that contains objects of type `T`, where `T` is either a primitive type or a `struct` type declared in the structure definition part of the specification. The set `S` has `n` subsets $\mathtt{S}_1, ..., \mathtt{S}_\mathtt{n}$ which together partition `S`. Changing the `partition` keyword to `subsets` removes the requirement that the subsets partition `S` but otherwise leaves the meaning of the declaration unchanged. A relation declaration of the form `relation R:` $\mathtt{S}_1$`->`$\mathtt{S}_2$ specifies a relation between the objects in the sets $\mathtt{S}_1$ and $\mathtt{S}_2$.

The model definition rules define a translation from the concrete data structures into an abstract model. Each rule has a quantifier that identifies the scope of the rule, a guard whose predicate must be true for the rule to apply, and an inclusion constraint that specifies an object (or tuple) that must be in a given set (or relation). Figure 7 presents the grammar for the model definition language.

$$
\begin{array}{rcl}
C & := & Q^*, G \Rightarrow I \\
Q & := & \texttt{for}\ V\ \texttt{in}\ S \mid \texttt{for}\ \langle V, V\rangle\ \texttt{in}\ R \mid \texttt{for}\ V = E\ ..\ E \\
G & := & G\ \texttt{and}\ G \mid G\ \texttt{or}\ G \mid !G \mid E = E \mid E < E \mid \texttt{true} \mid \\
 & & (G) \mid E\ \texttt{in}\ S \mid \langle E, E\rangle\ \texttt{in}\ R \\
I & := & E\ \texttt{in}\ S \mid \langle E, E\rangle\ \texttt{in}\ R \\
E & := & V \mid number \mid string \mid E.field \mid E.field[E] \mid \\
 & & E - E \mid E + E \mid E/E \mid E * E
\end{array}
$$

Figure 7: Model Definition Language

In principle, the presence of negation in the model definition language opens up the possibility of unsatisfiable model definition rules. We address this complication by requiring the set of model definition rules to have no cycles that go through rules with negated inclusion constraints in their guards.

3.3 The Constraint Language

Figure 8 presents the grammar for the model constraint language. Each constraint consists of a sequence of quantifiers $Q_1, ..., Q_n$ followed by body B. The body uses logical connectives (and, or, not) to combine basic propositions P that constrain the sets and relations in the model. Developers use this language to express the key consistency constraints.

$$
\begin{array}{rcl}
C & := & Q, C \mid B \\
Q & := & \texttt{for}\ V\ \texttt{in}\ S \mid \texttt{for}\ \langle V, V\rangle\ \texttt{in}\ R \mid \texttt{for}\ V = E\ ..\ E \\
B & := & B\ \texttt{and}\ B \mid B\ \texttt{or}\ B \mid !B \mid (B) \mid VE\ comp\ E \mid \\
 & & V\ \texttt{in}\ SE \mid \texttt{size}(SE)\ comp\ C \\
comp & := & = \mid < \mid <= \mid > \mid >= \\
VE & := & V.R \mid R.V \mid (VE) \mid VE.R \mid R.VE \\
E & := & V \mid number \mid string \mid E + E \mid E - E \mid E/E \mid \\
 & & E * E \mid E.R \mid \texttt{size}(SE) \mid (E) \\
SE & := & S \mid V.R \mid R.V
\end{array}
$$

Figure 8: Model Constraint Language

4. COMPILATION AND OPTIMIZATION

We implemented a compiler that processes Archie specifications to generate C code that implements a basic consistency checking algorithm. This algorithm first uses a work-list-based fixed point algorithm to construct the model, then evaluates the consistency constraints to detect any possible inconsistencies. Unfortunately, this straightforward compilation strategy generates checking algorithms that are too slow for our purposes. We therefore implemented the following optimizations.

4.1 Fixed Point Elimination

This optimization analyzes the model definition rules to replace, when possible, the fixed point computation with a more efficient data structure traversal. The compiler first performs a dependence analysis on the model definition rules to generate a dependence graph. This graph captures the dependences between rules which create sets and relations and the rules which use those sets and relations. Formally, the graph consists of a set of nodes N (one for each rule) and a set of edges E. There is an edge $E = \langle N_1, N_2\rangle$ from N_1 to N_2 if N_2 *uses* a set or relation that N_1 *defines*. A rule *uses* a set S (or a relation R) if the rule has a quantifier of the form `for` V `in` S (or of the form `for` $\langle V_1, V_2\rangle$ `in` R) or if the rule has a guard of the form E `in` S (or $\langle E_1, E_2\rangle$ `in` R). A rule *defines* a set S (or relation R) if it has an inclusion constraint I of the form E `in` S (or $\langle E_1, E_2\rangle$ `in` R).

[2]Note that the notation `CITYMAP.c` denotes the inverse image of `c` under the relation `CITYMAP` (the set of all `t` such that ⟨`t,c`⟩ in `CITYMAP`).

The compiler topologically sorts the strongly connected components in the dependence graph. For components that consist of a single rule, the compiler generates efficient code that iterates through all of the rule's possible quantifier bindings, evaluates the guard for each binding, and (if the guard is satisfied) executes the actions that add the appropriate objects to sets or tuples to relations. For components that consists of multiple rules, the compiler generates code that performs a fixed point computation of the sets and relations that the component produces. The generated code executes the computations for the components in the topological sort order. This order ensures that each set and relation is completely constructed before it is used to construct additional sets and relations in other components.

4.2 Relation Elimination

Some of the relations constructed in our model correspond to partial functions. For example, a field f may generate a relation that relates each object o to the value of the field $o.f$. Our compiler discovers relations that implement partial functions and verifies that these relations are used only in the forward direction (i.e., no expression uses the inverse of the relation). The compiler recognizes that a relation R is a partial function if the model definition rules use a single rule of the following form to define R:

$\texttt{for}\ V\ \texttt{in}\ S,\ G\ \Rightarrow\ \langle V, E\rangle\ \texttt{in}\ R.$

The compiler rewrites each expression that uses a partial function by replacing the use with the computation of G and (if G is satisfied) E. The compiler then removes the rule responsible for constructing each such relation.

4.3 Set Elimination

Our final optimization attempts to transform the specification to eliminate set construction and instead perform the checks directly on the data structures in memory. We use two transformations: *model definition rule inlining* and *constraint inlining*. Model definition rule inlining finds a model definition rule of the form $Q^*,\ G_1\ \Rightarrow\ V_1\ \texttt{in}\ S$, a second model definition rule of the form $\texttt{for}\ V_2\ \texttt{in}\ S,\ G_2 \Rightarrow I$, then eliminates the use of the set S in the second rule by transforming it to $Q^*,\ G_1 \wedge G_2[V_1/V_2] \Rightarrow I[V_1/V_2]$. To apply the transformation, the first rule must be the only rule that defines S.

The constraint inlining transformation finds a model definition rule of the form $Q^*,\ G\ \Rightarrow\ V_1\ \texttt{in}\ S$, a consistency constraint of the form $\texttt{for}\ V_2\ \texttt{in}\ S, C$, then eliminates a use of the set S by transforming the consistency constraint to $Q^*,\ G\ \Rightarrow C[V_1/V_2]$. To apply the transformation, the model definition rule must be the only rule that defines S. Note that the new constraint has a predicate ($G\ \Rightarrow\ C[V_1/V_2]$) that may involve both concrete values from the data structures in memory and the sets and relations in the model. We have extended the internal representation of our compiler so that it can generate code to check these kinds of hybrid constraints.

Each transformation eliminates a use of the set S. If the transformations eliminate all uses, the compiler removes the set and the rule that produces the set from the specification, eliminating the time and space required to compute and store the set. This optimization can be especially useful when (as is the case for our benchmark system) the compiler is able to eliminate the largest sets or relations.

4.4 Performance Impact

Table 1 presents the execution times of our benchmark software system with the consistency checks at different optimization levels. As these numbers show, the optimizations produce dramatic performance improvements. The final optimized version is more than efficient enough for interactive debugging use.

5. ENVISIONED USAGE STRATEGY

Obtaining developer acceptance of a new tool can be difficult, especially when the tool requires the developers to use a new language such as our specification language. We expect that several aspects of Archie will facilitate its acceptance within the developer community:

Version	Time
No Instrumentation	0.234 sec
Baseline Compiled	20 min
Fixed point elimination	25.60 sec
Relation Elimination	10.66 sec
Set removal	1.45 sec

Table 1: Performance Results

- **Black Box Usage:** The specifications can be developed by a small number of developers who are familiar with the specification language, while the remainder of the developers can simply use Archie as a black box. We anticipate no need for the vast majority of the developers to learn the Archie specification language. There is also no need to change the programming language, coding style, or other development tools.
- **Incremental Adoption:** Archie supports incremental adoption — the developer can start with a specification that captures a small subset of the consistency properties, then incrementally augment the specification to capture more properties. During the specification development process the consistency checker becomes more useful as more properties are added. Calls to Archie can also be incrementally added to the system. The overall result is a smooth integration into the development process with no major dislocations or disruptions.
- **Ease of Development:** Based on our experience developing similar specifications in another project [7], we believe that Archie specifications will prove to be relatively easy to develop once the developer understands the relevant data structures.[3] Because the specifications identify global data structure invariants rather than specific properties of local computations, our experience indicates that the resulting specifications are quite small (the largest are several hundred lines long, with the majority of the lines devoted to structure definitions) in comparison with the size of the software system as a whole.

We do anticipate that the use of Archie may wind up substantially changing the testing, error localization, and error correction activities, but in a positive way — we anticipate that Archie will help developers find errors earlier and provide them with substantially improved error localization. The developers in our case study (see Section 6) had no problem integrating Archie into their debugging strategy and in fact used Archie almost immediately to eliminate tedious activities such as augmenting the code with print statements or using a debugger to insert breakpoints and examine the values of selected variables.

We expect that Archie will effectively support usage strategies in which the initial specifications are developed as part of the software design process before coding begins and usage strategies in which it is integrated into a large existing software system. We also anticipate that, once integrated, the developers will be motivated to keep the specification up to date to reflect changes to the data structures. The division of the specification into model definition rules and consistency constraints facilitates this specification maintenance — if only the representation of the data changes, the developer can simply update the model definition rules to reflect the new representation, leaving the consistency constraints intact.

During development, we expect the program to be instrumented with calls to the Archie consistency checker. We anticipate two kinds

[3]Specifically, we have developed specifications for the FreeCiv interactive game, the CTAS air-traffic control system [1, 23] (this deployed system consists of over 1 million lines of code), a simplified version of the Linux ext2 file system [20], and Microsoft Word files.

of instrumentation: calls placed (potentially with the aid of an automatic call placement tool) at standard locations such as procedure entry and exit points as a routine part of the development process, and calls placed at chosen locations by developers as they attempt to localize a specific error.

6. CASE STUDY

Our case study attempts to answer the most basic question one could ask about Archie's potential effectiveness: Given a specification and a data structure corruption error that causes the data structures to violate this specification, does Archie help developers localize and correct the error? To answer this question, we obtained a benchmark software system and a population of developers, then performed a study in which the developers attempted to localize and correct errors in the system. By comparing the behavior and effectiveness of the developers that used Archie with the developers that did not, we are able to obtain an indication of how well Archie aided the error localization and correction process for this class of errors.

6.1 Developer Population

We recruited six developers with relatively homogeneous backgrounds: all developers had similar educational backgrounds, all represented their home country in international programming competitions while they were in high school, and all are currently students at MIT.

We separated the developers into two populations: the Tool population, which used Archie during the debugging experiments, and the NoTool population, which did not use Archie. To control for debugging ability, we assigned each developer a pre-study calibration task of locating and correcting an error in a heapsort implementation. We ordered the developers by the time required to correct this error; the times varied between 9 and 32 minutes. We then randomly assigned one of the first two, the next two, and the last two developers to the Tool population, with the others assigned to the NoTool population.

6.2 FreeCiv

We chose the FreeCiv interactive game program (available at `http://www.freeciv.org`) as our benchmark software system. The source code consists of roughly 65,000 lines of C in 142 files. It contains four modules: a server module, a client module, an AI module, and a common module. We have made all of the information required to replicate our results available at
`http://www.mit.edu/~cristic/Archie`.

6.2.1 Consistency Properties

FreeCiv maintains a map of tiles arranged as a rectangular grid. Each tile contains a terrain value (plains, hills, ocean, desert, etc.) and a reference to a bitmap which maintains additional information (such as pollution levels) about the tile. Each tile may also contain a reference to a city data structure. Our FreeCiv specification consists of 199 lines (of which 180 contain structure definitions). This specification identifies the following five consistency properties: each game must have a single map, each game must have a single grid of tiles, each tile must have a valid terrain value, exactly one tile must point to each city, and no city may be located on an ocean tile.

6.2.2 Incorrect Versions

We used manual fault insertion to create three incorrect versions of FreeCiv. The first version contains an error in the common module. The incorrect procedure is 14 lines long (after error insertion); the error causes the program to assign an invalid terrain value to a tile (causing the data structures to violate the third constraint identified above). The second version contains an error in the server module. The incorrect procedure is 18 lines long and causes two tiles to refer to the same city (causing the data structures to violate the fourth constraint). The third version also contains an error in the server module. The incorrect procedure is 153 lines long; the error causes a city to be placed on an ocean tile (violating the last constraint).

6.2.3 Experimental Setup

We first presented all of the developers with a FreeCiv tutorial, which gave them an overview of the purpose and structure of the program, an overview of Archie, and an overview of the FreeCiv data structures and their consistency properties.

We gave both the Tool and NoTool populations identical instrumented copies of the three incorrect versions of FreeCiv. These copies contain calls to the Archie consistency checker at the beginning and end of each procedure, with the exception of small procedures like structure field getters and setters and I/O procedures that interface with the user or the network. For the NoTool population, these calls immediately return without performing any consistency checking; for the Tool population, each call uses the Archie specification to perform a complete consistency check. Consistent with the expected usage strategy in Section 5, the Tool developers used Archie as a black box — they simply compiled the pre-generated consistency checker into their executables.

The instrumented versions of FreeCiv contain approximately 750 statements that invoke the Archie consistency checker. For the Tool population, each call (whether it detects an inconsistency or not) writes an entry to a log indicating the position in the code from which it was invoked. For this study, we configured FreeCiv to use its autogame mode in which it plays against itself and set the random number generator seed to a fixed value (to ensure repeatability). In this mode, the correct version of the program invokes the checker more than 20,000 times when it executes.

We asked the developers to attempt to locate and eliminate the errors in the three incorrect versions. We requested that they spend at least one hour on each version and allowed them to spend more time if they desired. For the NoTool population, each error manifested itself as either an assertion violation (the first two errors) or a segmentation fault (the last error). For the Tool population, each error manifested itself as an error message from the Archie consistency checker — the consistency checker printed out the violated constraint, the location of the call to the consistency checker, and an explanation of the error provided by the developer of the specification.

All of the developers used a Linux workstation (RedHat 8.0 Linux) with two 2.8 GHz Pentium 4 processors and 2 GBytes of RAM. We provided all of the developers with scripts to compile and run the three versions. The developers were able to use any development or debugging tool available on this platform. The developers were all familiar with this computational environment and comfortable using it. We observed the developers during the experiment and maintained a detailed record of their actions.

6.3 The Tool Population

Table 2 presents the number of minutes required for each member of the Tool population to locate each error; Table 3 presents the total number of minutes required to both locate and correct the error. As these numbers show, the developers were able to locate and correct the errors quite rapidly.

The developers in this population used Archie extensively in their debugging activities. They all started by examining the Archie inconsistency message. If the message came from a call to the Archie consistency checker at the start of a procedure, they examined the Archie log to find the caller of this procedure and (correctly) attributed the error to the caller. If the message came from a call to the Archie consistency checker at the end of a procedure, they (once again correctly) attributed the error to this procedure.

Participant	Error 1	Error 2	Error 3
T1	1	2	1
T2	2	3	2
T3	5	1	5

Table 2: Localization Times (Tool)

Participant	Error 1	Error 2	Error 3
T1	9	7	3
T2	8	6	8
T3	17	7	14

Table 3: Correction Times (Tool)

They then examined the message to determine which constraint was violated, then examined the code of the procedure containing the error to find the code responsible for the inconsistency. For the third error (recall that the procedure containing this error is 153 lines long) the developers inserted additional calls to the Archie consistency checker to further narrow down the source of the inconsistency. Eventually all of the developers found and eliminated the error.

6.4 The NoTool Population

Table 4 presents the number of minutes required for each member of the NoTool population to locate each error; Table 5 presents the total number of minutes required to both locate and correct the error. A dash (-) indicates that the developers were unable to locate or correct the error; a number in parenthesis after the dash indicates the number of minutes spent on the respective task before giving up. As these tables indicate, only one of the developers was able to locate and correct an error. Moreover, this correction was somewhat fortuitous: the developer spent the last 15 minutes of his attempt to locate the second error examining the correct version of the procedure that was modified to contain the third error. When he reexamined this procedure during his attempt to locate the third error, he noticed that the code was different and simply replaced the incorrect version with the correct version that he had examined earlier!

Participant	Error 1	Error 2	Error 3
NT 1	-	-	10
NT 2	-	-	-
NT 3	-	-	-

Table 4: Localization Times (NoTool)

Participant	Error 1	Error 2	Error 3
NT 1	- (95)	- (65)	11
NT 2	- (90)	- (70)	- (60)
NT 3	- (70)	- (60)	- (60)

Table 5: Correction Times (NoTool)

For the first two versions of FreeCiv, the developers in the NoTool population started by examining the code that triggered the assert violation. For the third version, the developers started their examination with the code that triggered the segmentation fault. Once it became clear to them that the code surrounding the assertion or segmentation fault was not responsible for the inconsistency, they attempted to trace the execution backwards to locate the code responsible for the error. During this process, they made extensive use of gdb to set break points and examine the values of the program variables. They also inserted print statements to track the values of different variables and augmented the program with additional assertions to check various consistency properties. Our observations indicate that all of the developers in this group made meaningful progress towards localizing the error. But because of the complexity of the program and the time between the generation of the inconsistency and its manifestation, they were unable to successfully localize the error within the amount of time they were willing to spend.

After several days we asked the developers in the NoTool population to attempt to use Archie to localize and correct the errors. Tables 6 and 7 present the localization and correction times, respectively.[4] As these results show, once the NoTool developers were given access to Archie, they were able to quickly localize and correct the errors.

Participant	Error 1	Error 2	Error 3
NT 1	1	2	-
NT 2	3	2	1
NT 3	3	1	8

Table 6: Localization Times (NoTool with Archie)

Participant	Error 1	Error 2	Error 3
NT 1	2	3	-
NT 2	4	3	6
NT 3	4	3	19

Table 7: Correction Times (NoTool with Archie)

6.5 Discussion

Error localization was the crucial step for debugging the errors in our study and Archie's ability to detect and flag each inconsistency immediately after it was generated was primarily responsible for the divergent experiences of the two populations. Developers in both populations had a clear manifestation of the error and started the debugging process by examining the code that produced this manifestation. For the Tool population, Archie produced a manifestation that quickly directed each developer to the procedure containing the incorrect code. Once directed to this procedure, the developers were able to quickly and effectively locate and correct the error.

	Significant Procedure Calls	Execution Time (%)
Error 1	12689	15%
Error 2	579	1%
Error 3	4142	8.5%

Table 8: Error to Manifestation Distance

Without Archie, the program executed for a substantial period of time before the data structure inconsistency finally manifested itself as an assertion violation or segmentation fault. Table 8 presents numbers that quantify this distance. The first column presents the number of significant procedure calls (this number excludes getter, setter, and I/O procedure calls) between each error and its manifestation as an assertion violation or segmentation fault; the second column presents this distance as a percentage of the running time of the correct version.

Moreover, the inconsistency did not cause incorrect code to fail — it instead caused distant correct code to fail, misleadingly directing the developer to fruitlessly examine correct code instead of incorrect code as the source of the error. Even though the NoTool population was able to obtain a reasonably accurate understanding of each error, their inability to localize the error (even given their understanding) prevented them from correcting it. And once the NoTool population was given access to Archie, they were able to use Archie to quickly and effectively locate and correct the error.

[4]There are no results for developer NT 1 on error 3 because this developer localized and corrected this error in the previous experiment.

6.5.1 Comparison With Assertions

Our results reveal several limitations of assertions as a debugging tool. Like Archie, assertions test basic consistency constraints and, if a constraint is violated, tell the developer which property was violated and where in the execution the violation was detected. It is therefore not clear that Archie should provide any benefit for a program whose assertions successfully detect inconsistencies. But in our study, Archie proved to be substantially more useful to the developers than the assertions, *even though two out of the three data structure inconsistencies manifested themselves as assertion violations*. There are two (related) reasons for this (counterintuitive) result: 1) the assertions in FreeCiv detected the inconsistencies long after their generation, and 2) the assertions did not direct the developers to inconsistencies in the initially corrupted data structures — they instead directed them to inconsistencies in data structures derived from the initially corrupted data structures.

The assertions in FreeCiv, as in many other programs, tend to test easily available values accessed by the surrounding code. The assertions therefore test only partial, local properties of the accessed parts of the data structure, typically properties that the code containing the assertion relies on for its correct execution. In particular, if a computation reads some data structures and produces others, the assertions tend to test the read data structures, not the produced data structures.

It is therefore possible (and even likely) for a program to execute successfully through many assertions after it corrupts its data structures. And when an assertion finally catches the inconsistency, the execution may be very far away from the code responsible for the inconsistency and the inconsistency may have propagated through additional data structures. In our incorrect versions of FreeCiv, for example, one phase of the program produces an inconsistent data structure, but the assertions detect these inconsistencies only after a distant phase attempts to read a data structure derived from the original inconsistent data structure — the intervening phases either do not attempt to access this data structure or fail to check for the violated consistency property.

Because Archie comprehensively checks all of the consistency properties, it makes the developer aware of the inconsistency as soon as it occurs. This immediate notification was crucial to its success in our study, because (unlike the delayed notification characteristic of the existing FreeCiv assertions) it immediately directed the developers to the incorrect code and identified the data structure that it corrupted (and not some other derived data structure).

6.5.2 Efficiency

The basic benefit of Archie is to localize each error to the region of the execution between the failed consistency check and the immediately preceding successful consistency check. It is therefore desirable to perform the consistency checks as frequently as possible so as to better localize the error. The primary obstacle to frequent consistency checking is the overhead of executing the checks.

The optimizations discussed in Section 4 are therefore crucial to the successful use of Archie. Without optimization, the consistency checks increase the FreeCiv execution time from less than a second to twenty minutes. While this kind of time dilation may be acceptable for errors that would otherwise be very difficult to localize, we would prefer to enable developers to use Archie routinely during all of their executions.

Our optimizations enabled us to provide the developers in our study with a checker that can execute frequently while maintaining an interactive debugging environment. We believe that this level of efficiency was crucial to the successful use of Archie in our study and that our optimizations will prove to be at least as important for obtaining an acceptable combination of check frequency and response time for other applications.

6.5.3 Applicability

Our study indicates that consistency checking in general and Archie in particular can help developers locate and eliminate data structure corruption errors that violate the checked consistency property. For this class of errors, Archie provides the developer with information that helps the developer to both localize the bug and understand the violated consistency properties.

We believe that Archie is less likely to be useful for finding errors that do not result in data structure corruption, although it could still be useful for ruling out classes of errors. However, from our experiences we believe that data structure corruption errors are a particularly difficult class to debug, and that Archie should prove useful in practice for this class of errors.

6.5.4 Future Work

This study leaves many interesting questions unanswered. In particular, it provides no indication whether a specification-based approach provides any advantages or suffers from any disadvantages as compared with an approach based on manually developing consistency checkers in the standard implementation language. We anticipate that in either case an expert would develop the specification or consistency checker, most developers would use the consistency checker as a black box, and the development of the consistency checker would require a small fraction of the overall development time. Potential advantages of the specification-based approach include reduced development time, a clearer and more explicit identification of the important consistency properties, and consistency checkers with fewer errors that are amenable to static analysis and targeted optimizations (such as those discussed in Section 4). It remains to be seen if these potential advantages materialize in practice.

A second area of potential inquiry concerns the frequency, relative importance, and consistency violation properties of data structure corruption errors in practice. Our study leaves open questions of whether data structure corruption errors are an important problem in practice and whether developers are able to produce specifications or consistency checkers that catch the kinds of data structure corruptions that occur in practice.

7. RELATED WORK

Error localization and correction has been an important issue ever since people began to develop software. Researchers have developed a host of dynamic and static debugging tools; a small selection of recent systems includes [9, 4, 25, 11, 2, 5, 26, 15, 16, 8]. We confine our survey of related work to research in specification languages, specification-based testing, and invariant inference systems.

7.1 Specification Languages

The basic concepts in our specification language (objects and relations) are the same as in object modeling languages such as UML [22] and Alloy [13], and the specification language itself has many of the same concepts and constructs as the constraint languages for these object modeling languages, which are designed, in part, to be easy for developers to use.

Standard object modeling approaches have traditionally been used to help developers express and explore high-level design properties. One of the potential benefits of our approach is that it may enable developers establish a checked connection between the high-level concepts in the model and their low-level realization in the data structures in the program.

7.2 Specification-Based Testing

Specification-based testing (of which Archie is an instance) tests the correctness of an execution by determining if it satisfies a specification written in some specification language. Specification-based testing is usually implemented at the granularity of procedure preconditions and postconditions. ADL [24], JML [14], Testera [18], Korat [3], and several Eiffel [19] implementations, to name a few, implement various forms of this kind of specification-based testing.

Archie, in contrast, implements a global invariant checker with no attempt to verify any property of the execution other than the preservation of the invariant. Advantages of Archie include reduced specification overhead and complete coverage of the global invariants (instead of checking more targeted properties that are intended to characterize procedure executions); the disadvantage is that it is not intended to find errors that do not violate the invariant. Our evaluation is that the two kinds of checkers address complementary properties and that both provide valuable checking functionality.

7.3 Invariant Inference and Checking

Several research groups have developed systems that dynamically infer likely invariants or other program properties; the same technology can be easily used to check the inferred properties (or, for that matter, any property expressed using the same formalism). Specific systems include DAIKON [10], Carrot [21], DIDUCE [12], and automatic role inference [6].

An important difference between Archie and these previously existing systems is that Archie is designed to check the substantially more sophisticated properties characteristic of complex linked data structures that must satisfy important structural constraints. The (in our view minimal) overhead is the need to provide a specification of these properties instead of automatically inferring the properties. And in fact, it would be feasible to use automatic property discovery tools to generate Archie consistency constraints or to obtain an initial set of properties that could be refined to obtain a more precise specification.

8. CONCLUSION

Error localization is a necessary prerequisite for correcting software errors and often the primary obstacle. Archie addresses this problem by accepting a specification of key data structure consistency properties, then automatically checking that the data structures satisfy these properties. The Archie checker can help developers quickly localize data structure corruption errors to the region of the execution between two subsequent calls to Archie.

Our set of optimizations enables the Archie compiler to generate checking code that executes more than efficiently enough to enable an effective check frequency and support its routine use in an interactive debugging environment. Moreover, the results from our case study indicate that developers can almost immediately use Archie to substantially improve their ability to localize and correct errors in a substantial software system. We believe that Archie therefore holds out the potential to substantially improve the ability of developers to first localize, then correct, data structure corruption errors.

Acknowledgements

This research was supported in part by DARPA Contract F33615-00-C-1692, NSF Grant CCR00-86154, NSF Grant CCR00-63513, NSF Grant CCR00-73513, NSF Grant CCR-0209075, NSF Grant CCR-0341620, and NSF Grant CCR-0325283,

9. REFERENCES

[1] Center-tracon automation system. `http://www.ctas.arc.nasa.gov/` .

[2] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. *Lecture Notes in Computer Science*, 2057:103+, 2001.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, July 2002.

[4] J.-D. Choi et al. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the SIGPLAN '02 Conference on Program Language Design and Implementation*, 2002.

[5] M. Das, S. Lerner, and M. Seigle. Path-sensitive program verification in polynomial time, 2002.

[6] B. Demsky and M. Rinard. Role-based exploration of object-oriented programs. In *ICSE02*, May 2002.

[7] B. Demsky and M. C. Rinard. Automatic detection and repair of errors in data structures. In *OOPSLA*, October 2003.

[8] M. Ducass. Coca: An automated debugger for c. In *Proceedings of the 21st International Conference on Software Engineering*, 1999.

[9] D. Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *SOSP*, October 2003.

[10] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224, 1999.

[11] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *PLDI*, pages 69–82, 2002.

[12] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, May 2002.

[13] D. Jackson. Alloy: A lightweight object modelling notation. Technical Report 797, Laboratory for Computer Science, Massachusetts Institute of Technology, 2000.

[14] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, Iowa State University, 2000.

[15] R. Lencevicius, U. Hlzle, and A. K. Singh. Query-based debugging of object-oriented programs. In *OOPSLA97*, October 1997.

[16] B. Lewis. Debugging backwards in time. In *Proceedings of the Fifth International Workshop on Automated Debugging AADEBUG 2003*, 2002.

[17] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[18] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE)*, Nov. 2001.

[19] B. Meyer. *Eiffel: The Language*. Prentice Hall, New York, NY, 1992.

[20] D. Poirier. Second extended file system. `http://www.nongnu.org/ext2-doc/` , Aug 2002.

[21] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss. Automated fault localization using potential invariants. In *Proceedings of the 5th International Workshop on Automated and Algorithmic Debugging*, September 2003.

[22] Rational Inc. The unified modeling language. `http://www.rational.com/uml`.

[23] B. D. Sanford et al. Center/tracon automation system: Development and evaluation in the field. In *38th Annual Air Traffic Control Association Conference Proceedings*, October 1993.

[24] S. Sankar and R. Hayes. Specifying and testing software components using ADL. Technical Report TR-94-23, Sun Microsystems, 1994.

[25] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[26] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the tenth ACM SIGSOFT symposium on Foundations of software engineering*, 2002.

Run Time Monitoring of Reactive System Models

Mikhail Auguston
Naval Postgraduate School
Monterey, CA, USA
auguston@cs.nps.navy.mil

Mark Trakhtenbrot
Academic Institute of Technology
Holon, Israel
ilmarktr@yahoo.com

Abstract

In model-based development of reactive systems, statecharts are widely used for formal design of system behavior, and provide a sound basis for analysis and verification tools, as well as for code generation from system models. We present an approach for dynamic analysis of reactive systems via run-time verification of code produced with Statemate C and MicroC code generators [10], [15]. The core of the approach is automatic creation of monitoring statecharts from formulas that specify the system's behavioral properties in a proposed assertion language. Such monitors are then translated into code together with the system model, and executed concurrently with the system code. This approach leads to a more realistic analysis of reactive systems, as monitoring is supported in the system's actual operating environment. For models that include design-level attributes (division into tasks, etc.), this is crucial for performance-related checks, and helps to overcome restrictions inherent in simulation and model checking.

1. Introduction

Development of reliable reactive systems is a significant challenge, especially due to their complex behavior. There has been a great deal of research on the development of formal methods for specification, design, analysis and verification of reactive systems.

For precise specification of system behavioral properties, various types of temporal logic are widely used. These include LTL [14], which offers special temporal operators for reasoning about past and future properties of behavioral sequences, and MTL [5], which supports expression of real-time constraints through definition of duration for future temporal operators. Some specification formalisms suggest various kinds of syntax sugar that make the specification task more user friendly for designers who are not logicians. For example, with the LA language in [18], temporal properties look as a combination of stylized English with C-like expressions. In [3], the temporal logic details are hidden "behind the scenes", and instead, patterns are used that allow to specify common properties (such as existence, absence, response, precedence, etc.) and scope in which the property should hold. This approach is used, for example, in a Statemate verification tool called ModelCertifier [16] that offers a rich library of pre-defined property patterns, where each pattern looks as a parameterized natural language sentence. Paper [6] introduces a language for pattern definition as a way to create extendable sets of property patterns. Sugar [19] provides several layers for property specification and verification; in particular, extended regular expressions are used to describe execution sequences on which temporal properties are checked.

On the other hand, model-based system development has become the way to design, implement and validate reactive systems. Statecharts, first introduced in [9], have become a standard for behavior design in popular model-based methodologies such as structured and object-oriented design [7]. Various tools (e.g., Statemate [10], Rhapsody [9], BetterState [20]) support the creation of executable models using statecharts, and their analysis through simulation, execution of automatically generated code, and, in Statemate, verification. Ongoing research on model-based testing covers, among other issues, test generation from statechart models [4].

One powerful method of dynamic analysis is run-time monitoring of system execution. A number of tools have been developed for monitoring various types of programs (including real-time systems); see, for example [1], [2], [18]. The relevant assertion languages allow for expressing a wide range of properties in terms of events that occur in the running code, and for defining tool reactions when a violation is found or when the run was successful. An important problem here is the gap between the system specification, which usually refers to high-level objects, and monitors, which refer to implementation-level events (such as function calls, etc.). Some issues related to derivation of monitors from system specification are considered in [17].

Model-based development leads to a narrowing of this gap, as monitoring can be performed on the model (rather than the implementation) level. Statemate [10] supports the use of the so-called watchdog (testbench) statechart. Such a chart is not part of the system model; its role is either that of a driver (acting as an environment and producing system inputs) or a monitor (watching the system for proper behavior or abnormalities). To perform its role, the watchdog is executed in parallel with the model. Violation of the monitored property can be expressed and observed as entering an error state in the monitor chart. For example, Fig. 1 shows a simple statechart for monitoring the following requirement: "Processing of a request must be accomplished within 5 seconds, and before receiving the next request".

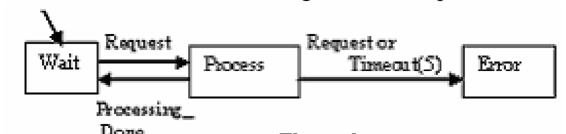


Figure 1

An important feature of monitor statecharts is that they have access to all elements in the system model. In other words, visibility from the monitor is supported both for observable elements (events, conditions and data items) that belong to system's interface with the environment, and for internal elements such as states or events used for internal communication between system components. This allows for both black box and more detailed white box monitoring, and makes localization of design problems easier.

2. What is in this paper

This paper presents an approach to dynamic analysis of reactive systems modeled with statecharts using Statemate. The basic goal here is to reveal errors (rather than to validate or show correctness).

The analysis is based on run-time monitoring of code generated from the system model. The code is checked against the system specification describing the required and forbidden behaviors; these are expressed in a proposed assertion language described below. The main idea underlying this approach is the automatic creation of monitors directly from the system specification. This is achieved through translation of the specification into an equivalent watchdog statechart(s). This step is followed by generating code from the system model and from the created monitor (using the existing Statemate C code generator), and their simultaneous execution. Appropriate diagnostics is produced during the execution and/or upon its completion.

The suggested approach has a number of advantages, and is especially helpful in situations where the use of other analysis tools (e.g. of model checkers such as Statemate ModelCertifier [16]) becomes problematic:

- There is no restriction on the size of the tested model, and execution of compiled code (for model and monitor) is fast. On the other hand, model checking may become slow for very large real-world models.

- Generated code for the system and its monitor is executed in ***real time***. Even though such code is usually considered prototype quality, it is fast enough and allows for meaningful checks of time constraints (unless they are tighter than the code performance). Such checks are beyond the scope of simulation and model checking tools that are based on simulated time schemes described in [12]: synchronous (for clock-driven systems) and asynchronous (for event-driven systems). In the synchronous scheme duration of all steps is the same, regardless of how "heavy" the executed actions are. In the asynchronous scheme, steps take zero time, and the system executes a chain of steps until stabilization; only then is the clock advanced and inputs accepted. These abstractions are based on the assumption that the system is fast enough to complete its reactions to external stimuli before the next stimulus arrives. Real time monitoring allows one to check whether this assumption is correct.

- Our approach allows monitoring of code generated from the Statemate model augmented by design attributes (showing the system division into tasks of various types, mapping model elements into events of the target RTOS, etc.). For such models, the MicroC code generator [15] automatically creates a highly optimized production quality code for the OSEK operating system, widely used in the automotive industry for embedded microcontroller development. Thus the code can be executed and monitored in its realistic hardware-in-the-loop operating environment. This kind of analysis is impossible with model checking.

- Model checking requires that all data be properly restricted, to guarantee that a finite state model is analyzed. This requirement is problematic for input data, if there is not enough information about the system environment. No such restrictions are relevant for monitoring, and moreover, monitored code derived from the system model can be connected to real sources of input data.

3. Assertion language

To specify and monitor real-time properties of reactive systems, we use an assertion language that integrates a number of powerful features found in temporal logic and in FORMAN language (the latter was introduced in [1], [2], and is used in a number of tools):

- Boolean expressions can refer to any elements in the system model, and express properties of system configurations. For example: *in(S) and (x>5)* means that currently the system is in state *S* and *x* is greater than 5.

- Regular expressions allow for description of state sequences. Consider for example, the expression:

[SELECT (Open | Read | Write | Close) FROM ex_program] SATISFY Open (Read | Write) Close*

This assertion requires to select execution trace states matching one of the given patterns, and to check the sequence of selected states for conformance with the regular expression.

- Temporal formulas express order properties fulfilled by system execution sequences. They are built using unrestricted future temporal operators *NEXT, ALWAYS, EVENTUALLY, UNTIL* and their past counterparts: *PREVIOUS, ALWAYS_WAS, SOMETIME_WAS, SINCE.* Following [14], we consider formulas for the following types of properties (where *P* is a past formula):

Safety: *ALWAYS (P)*
Guarantee: *EVENTUALLY (P)*
Obligation: Boolean combination of safety and guarantee
Response: *ALWAYS (EVENTUALLY(P))*
Persistence: *EVENTUALLY (ALWAYS(P))*
Reactivity: Boolean combination of response and persistence.

According to [14], any temporal formula is equivalent to a reactivity formula; the other five types of formulas are allowed for more flexibility. For convenient expression of real-time constraints, we support also a restricted version of the above operators; it is obtained by attaching appropriate time characteristics. For example, *ALWAYS(10)P* means that *P* is continuously true during 10 time units after the current moment, while *SOMETIME_WAS (10) P* denotes that *P* was true at least once in the 10 previous time moments. With this extension, *P* in the above formulas is now allowed to be a restricted (future or past) formula. Note that we don't allow an unrestricted temporal operator to be nested within a restricted one.

- Actions define what should be done when a property violation is found, or when the property holds for the checked run. Typically, this includes sending an appropriate message. In general, any user-defined functions can be used here to provide a meaningful report that may include, for example, interesting statistics and other profiling information (frequency of occurrence for certain event, total time spent by the system in certain state, etc.). For this, actions can use the appropriate attributes of the referred objects (e.g., the time at which a certain interval was entered).

The examples in section 4 illustrate the use of this assertion language. Since the language is based on constructs described elsewhere (see [14], [12] and [1]), detailed description of its syntax and semantics is omitted from this paper. Nevertheless, one delicate issue should be mentioned here. System specification usually assumes infinite execution sequences (as a reactive system has an ongoing interaction with its environment). Correspondingly, the traditional semantics of temporal operators is also defined for infinite execution sequences. However, monitoring usually deals with finite (truncated) runs, and this requires a proper definition of the semantics for cases when there is doubt as to what would have been the property formula value if the execution had not been stopped. Paper [7] studies several ways of reasoning with temporal logic on truncated executions. We follow the so called neutral view discussed in [7]; this is illustrated by the following example. Consider the assertions:

ALWAYS (P → EVENTUALLY (10) Q)
ALWAYS (P → ALWAYS (10) Q)

and suppose that the run is completed (truncated) 4 seconds after the last occurrence of event *P* (we assume that each of the properties held for all earlier occurrences of *P*). If there was no *Q* after the last *P*, then the first assertion is considered to be false for this run (even though continuation of the run could reveal that *Q* does occur in 10 seconds after *P*, as required). On the contrary, if *Q* held continuously after the last *P* and until the end of the run, then the second assertion is considered to be true. In general, it is the user's responsibility to make the on-satisfy and on-failure actions detailed enough, so that he can better understand the monitoring results (e.g. whether a real violation was found, or it is in doubt due to the state at which the execution was truncated).

4. Examples

To illustrate our approach, we consider the Early Warning System (EWS) example from [12]. We present its verbal description followed by the statechart presenting the behavioral design of the system. We then give examples of assertions and, for one of them, show its translation into a monitor statechart according to our translation scheme.

The EWS receives a signal from an external source. When the sensor is connected, the EWS performs signal sampling every 5 seconds; it processes the sampled signal and checks whether the resulting value is within a specified range. If the value is out of range, the system issues a warning message on the operator display. If the operator does not respond to this warning within a given time interval (15 seconds), the system prints a fault message and stops monitoring the signal. The range limits are set by the operator. The system is ready to start monitoring the signal only after the range limits are set. The limits can be redefined after an out-of-range situation has been detected, or after the operator has deliberately stopped the monitoring.

Fig. 2 shows a statechart describing the EWS, similar to the one in [12]. The main part of EWS behavior is detailed in state *ON*. It contains two *AND*-components that represent the EWS controller and the sensor acting concurrently. Events *DO_SET_UP, EXECUTE,* and *RESET* represent the commands that can be issued by the operator. Timing requirements are represented by delays that trigger the corresponding transitions. The *AND*-components can communicate; for example, see event *CONNECT_OFF* sent from the controller component to the sensor component.

Following are four examples of assertions that reflect some of the above requirements for EWS:

1) *ALWAYS (EXECUTE → SOMETIME_WAS (DO_SET_UP))*

(monitoring of signal should be preceded by setting range limits)

2) *ALWAYS (OUT_OF_RANGE →*
   EVENTUALLY (15) (RESET or started(PRINT_ALARM))

(in the out-of-range situation, within 15 seconds either the operator responds or a fault message is printed)

3) *ALWAYS (*
   ALWAYS_WAS (15) (in(DISPLAY_ALARM) & not RESET)
   → started(PRINT_ALARM))

(a similar property, this time expressed using the past temporal operator)

4) *ALWAYS (FINISHED_SAMPLING →*
   ALWAYS (5) in(IDLE) or EVENTUALLY(5)CONNECT_OFF)

(after signal sampling is finished, there is a 5-second pause before the next sampling, unless the sensor is disconnected)

Note that the first assertion is violated for the given statechart; this happens in the following scenario: *POWER_ON; CONNECT_ON; EXECUTE*. The other assertions are valid as long as the system remains in its *ON* state (i.e., *POWER_OFF* doesn't occur), but otherwise can be violated.

Fig. 3 shows how the second of these four assertions is translated into a monitor statechart. Suppose *POWER_OFF* occurs 7 seconds after *OUT_OF_RANGE*, and there was no *RESET* in this interval. If the system remains in state *OFF* for at least the following 8 seconds, then the monitor will enter its state *D*, thus indicating a violation of the monitored assertion.

5. Implementation Outline

Statemate Boolean expressions obtained from basic predicates (like *in(DISPLAY_ALARM))*, guarding conditions, and event occurrences are directly visible from monitor statechart; in this sense, their monitoring is trivial. In monitors created to watch temporal and timing properties, such expressions can be used as transition triggers, similar to the example in Fig.1.

In the rest of this section, we present an outline of a translation scheme for restricted and unrestricted temporal formulas allowed by our assertion language (see section 3 above). Though not fully formalized here, the presentation clearly shows the technique used for generation of monitors from assertions.

Let *P, Q, S* denote basic Boolean formulas, which do not contain any temporal operators, and let *FRM* denote any formula.

Then $P \rightarrow Q$ means that *P* is used as a trigger to start monitoring of formula *Q*; for each occurrence of *P*, a new thread of *Q* monitoring should be started. Absence of the trigger (*P*→ ...) means that the start of execution is the only trigger event.

If a formula includes only restricted future temporal operators, like in

$$FRM \equiv P \rightarrow TL\_Operator\ (N1)\ TL\_Operator\ (N2)\ \dots\ TL\_Operator\ (Nk)\ S$$

then its value becomes known after (i.e. it needs to be monitored during), at most, $t(FRM) = N1 + N2 + \dots + Nk$ time units from the triggering event P. For example:

$$P \rightarrow ALWAYS(5)\ EVENTUALLY(10)\ S$$

is monitored during, at most, 15 time units from the triggering event *P*. For each step within the monitoring interval we have to know the Boolean values of all basic sub-formulas in the *FRM*. This is sufficient to determine, after *t(FRM)* time units, whether *FRM* is true or false for the particular occurrence of the trigger event *P*.

Every restricted future formula is translated into a chart containing two designated states: accepting state *F*, and rejecting state *D*; there are no transitions exiting from *F* and *D* in such a chart. The value of the formula is true when computation ends in *F*, and false when it ends in *D*. If execution of the monitored system is truncated before completion of the formula computation, then (in the spirit of the neutral view as defined in [7]) the value is decided to be true for the *ALWAYS*-formula and false for the *EVENTUALLY*-formula.

As an illustration, Fig. 4 schematically shows the translation pattern for $FRM \equiv ALWAYS\ (N)\ P$, where *P* itself is either a basic or a restricted future formula. Translation is defined by structured induction, starting from the case when *P* is a basic formula. Note that each advance of the clock by one time unit causes a new thread of computation for *P* to be started. Each thread is represented in the chart by a separate *AND*-component; there are *N* such components. This number is known based on an analysis of the translated formula.

Fig. 5 shows a translation pattern for a safety assertion where the unrestricted operator *ALWAYS* is applied to the restricted formula *P* (the actual structure of state *P* in each thread is defined by translation rules for restricted formulas). In this case, as long as *P* holds the value true, we should continue the ongoing computation of *P*. Whenever the monitor enters its *D* state, the value of the formula becomes false; otherwise (including the case of truncated execution), the value is true. Note that since obtaining a value of *P* may require up to *t(P)* time units, there are *t(P)* threads computing *P*. When a cycle of *P* computation is completed with the value true (the component reaches its *F* state), it is restarted again. Also note the delays: *RESTART_P_i* is defined in such a way that with each advance of the clock by one time unit, a new cycle of *P* computation is started. Restarting *P* immediately upon its completion in state *F* would have caused a violation of such synchronization in case a certain cycle takes less time than *t(P)*. This, in turn, could lead to wrong computation of the entire formula.

To implement *EVENTUALLY (ALWAYS(P))*, we have to restart computation of *ALWAYS(P)* whenever it gets the value false, i.e., when the chart in Fig. 5 enters state *D* (at the top level of the hierarchy). In other words, such implementation can be obtained by redirecting the transition from *D* back to the *AND*-state.

Implementation of dual formulas (where *ALWAYS* is replaced by *EVENTUALLY* and vice versa) is similar to the described above, with appropriate replacement of *F*-states by *D*-states and vice versa.

For restricted past formulas we need to monitor only the finite segment of the execution in order to decide whether the formula is true or false. Consider, for example, *ALWAYS_WAS (N) P* which means "during *N* time units preceding the current moment, *P* was continuously true". Implementation uses a counter *CP* associated with the formula; on each advance of the clock, if *P* is true then *CP* is incremented, and if *P* is false then *CP* is set to 0. Now *ALWAYS_WAS (N) P* is true at the current moment, iff *CP=N*.

Similarly, for *SOMETIME_WAS (N) P* that means "from the current moment in at least *N* previous steps *P* was true at least once", the implementation will use the counter *CP* in the following way: On each advance of the clock, if *P* is true then *CP* is set to *N,* and if *P* is false then *CP* is decremented by 1. Now, *SOMETIME_WAS (N) P* is true at the current moment, iff *CP > 0* at the current moment.

6. Conclusions and future work

The paper presents an approach to dynamic analysis of reactive systems via run-time verification of code generated from Statemate models. The approach is based on the automatic creation of monitoring statecharts from formulas that specify the system's temporal and real-time properties in a proposed assertion language. The promising advantage of this approach is in its ability to analyze realistic models (with attributes reflecting the various design decisions) in the system's realistic environment. This capability is beyond the scope of simulation and model checking tools.

Several experiments have been carried out, that included manual creation of monitor charts from assertion formulas and their use with C code generated from Statemate models (EWS considered in section 4, and some others). This helped in a more accurate definition of the translation scheme.

The natural next step is actual implementation of the translation from the assertion language into statechart monitors, which is the core of the suggested approach, and use of created monitors with real-world system models.

The assertion language needs to be more convenient for designers. A possible way to achieve this is to adopt some of the ideas discussed in [3], [6], [18], [19]. This will require an appropriate adaptation of the translation scheme.

The system described above for statechart run time monitoring is under development. The suggested translation scheme provides a uniform mechanism for automatic creation of monitors, although some examples show that, in certain cases, more compact and optimized monitors can be produced. Further research is needed to define a more efficient translation scheme, both for synchronous and asynchronous time models.

Finally, an interesting challenge is to check a similar approach with a UML-based design paradigm that uses an OO version of statecharts for behavior description. Here an additional advantage could be in monitoring of systems where objects are created dynamically such that their amount is not limited in advance (model checking analysis of such systems is clearly problematic).

7. Acknowledgements

This work has been supported in part by the U.S. Office of Naval Research Grant # N00014-01-1-0746.

8. References

[1] M. Auguston, Program Behavior Model Based on Event Grammar and its Application for Debugging Automation, *2nd Int'l Workshop on Automated and Algorithmic Debugging, AADEBUG'95*, May 1995, pp. 277-291.

[2] M. Auguston, A. Gates, M. Lujan, Defining a Program Behavior Model for Dynamic Analyzers, *9th International Conference on Software Engineering and Knowledge Engineering, SEKE'97*, June 1997, pp. 257-262.

[3] G.S. Avrunin, J. C. Corbett, and M. B. Dwyer, Property Specification Patterns for Finite-State Verification, *2nd Workshop on Formal Methods in Software Practice*, March 1998, pp.7-15.

[4] K.Bogdanov, M.Holcombe, H.Singh. Automated Test Set Generation for Statecharts. In D. Hutter, W. Stephan, P. Traverso and M. Ullmann, editors, *Applied Formal Methods - FM-Trends 98*, LNCS, v.1641, Springer Verlag, 1999, pp. 107-121.

[5] E.S. Chang, Z. Manna, and A. Pnueli. Compositional Verification of Real-time Systems. In *Proceedings of the 9th IEEE Symposium Logic in Computer Science (LICS 1994)*, IEEE Computer Society Press, 1994, pp. 458-465.

[6] J.C. Corbett, M.B. Dwyer, J. Hatcliff, Robby. A Language Framework for Expressing Checkable Properties of Dynamic Software. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, LNCS, v.1885, Springer-Verlag, 2000, p.205-223.

[7] B. P. Douglass, D. Harel and M. Trakhtenbrot. Statecharts in Use: Structured Analysis and Object-Orientation. *Lectures on Embedded Systems* (F. Vaandrager and G. Rozenberg, eds.), LNCS, v.1494, Springer-Verlag, 1998, pp. 368-394.

[8] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, D. Van Campenhout. Reasoning with Temporal Logic on Truncated Paths, In *Proceedings of 15th Computer-Aided Verification conference* (*CAV'O3*), LNCS, v.2725, Springer-Verlag , July 2003, pp.27-39,

[9] E. Gery, D. Harel and E. Palatchi. Rhapsody: A Complete Lifecycle Model-Based Development System, In *Proc. 3rd Int. Conference on Integrated Formal Methods*, IFM 2002, pp.1-10.

[10] D.Harel. Statecharts: A Visual Formalism for Complex Systems, *Science of Computer Programming*, 8, 1987, pp. 231-274.

[11] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtul-Trauring and M. Trakhtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems, *IEEE Trans. on Software Engineering* 16:4 (1990), pp.403-414.

[12] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Trans. on Software Engineering Method*. 5:4 (1996), pp.293-333.

[13] D.Harel, and M.Politi, *Modeling Reactive Systems with Statecharts: The STATEMATE Approach* McGraw-Hill, 1998

[14] Z.Manna and A.Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1991.

[15] M.Thanne and R.Yerushalmi. Experience with an Advanced Design Flow with OSEK Compliant Code Generation for Automotive ECU's. *Dedicated Systems Magazine*, *Special Issue on Development Methodologies & Tools*, pp. 6-11, 2001

[16] OSC – Embedded Systems AG. *Statemate ModelCertifier*. http://www.osc-es.de/products/en/modelcertifier.php

[17] D.Richardson, S.Leif Aha, T.Owen O'Malley. Specification-based Test Oracles for Reactive Systems, In *Proc. Fourteens Intl. Conf. on Software Engineering*, Melbourne, 1992, pp.105-118.

[18] O.Strichman, R.Goldring. The 'Logic Assurance (LA)' System - A Tool for Testing and Controlling Real-Time Systems, In *Proceedings of the 8th Israeli Conference on Computer Systems and Software Engineering,* 1997, pp.47-56.

[19] I.Beer, S.Ben-David, C.Eisner, D.Fisman, A. Gringauze and Y.Rodeh. The Temporal Logic Sugar. In *Intl. Conference on Computer Aided Verification* (*CAV'01*), LNCS, v.2102, July 2001, pp.363-367.

[20] Wind River Systems, Inc. *BetterState* http://www.windriver.com/products/betterstate/index.html

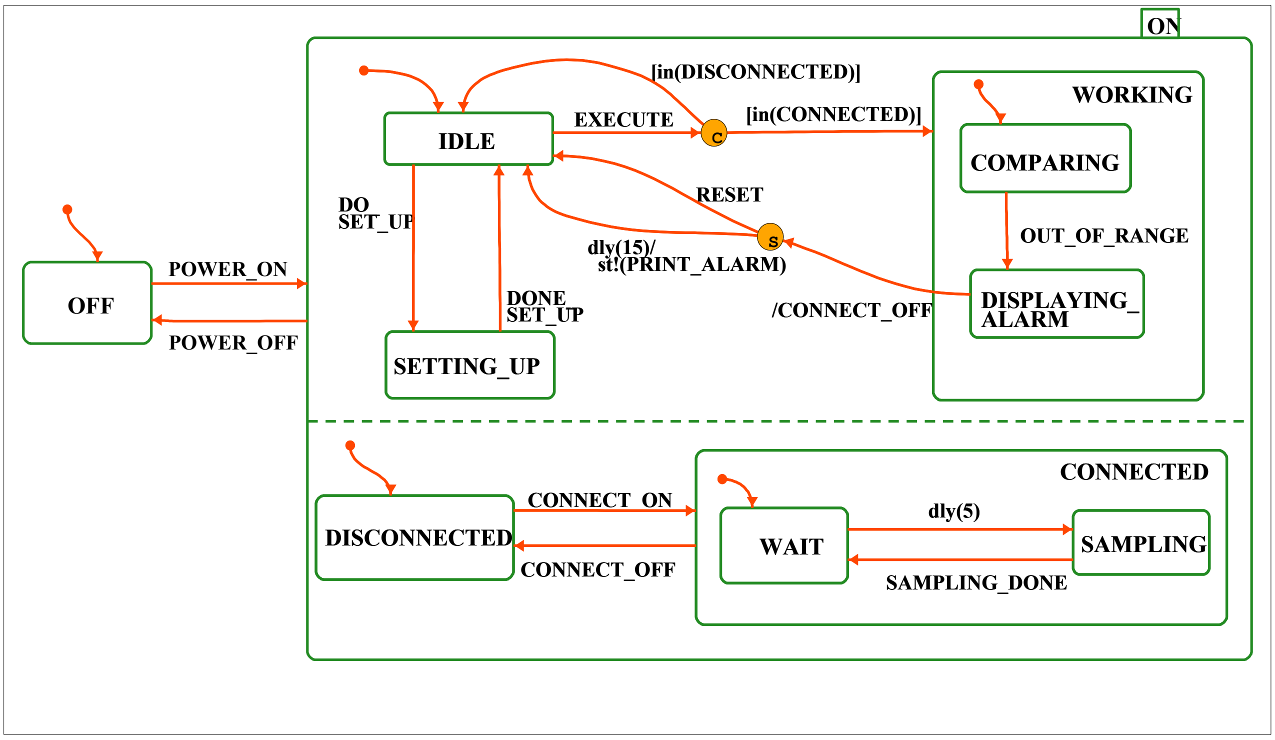


Figure 2. Statechart for Early Warning System

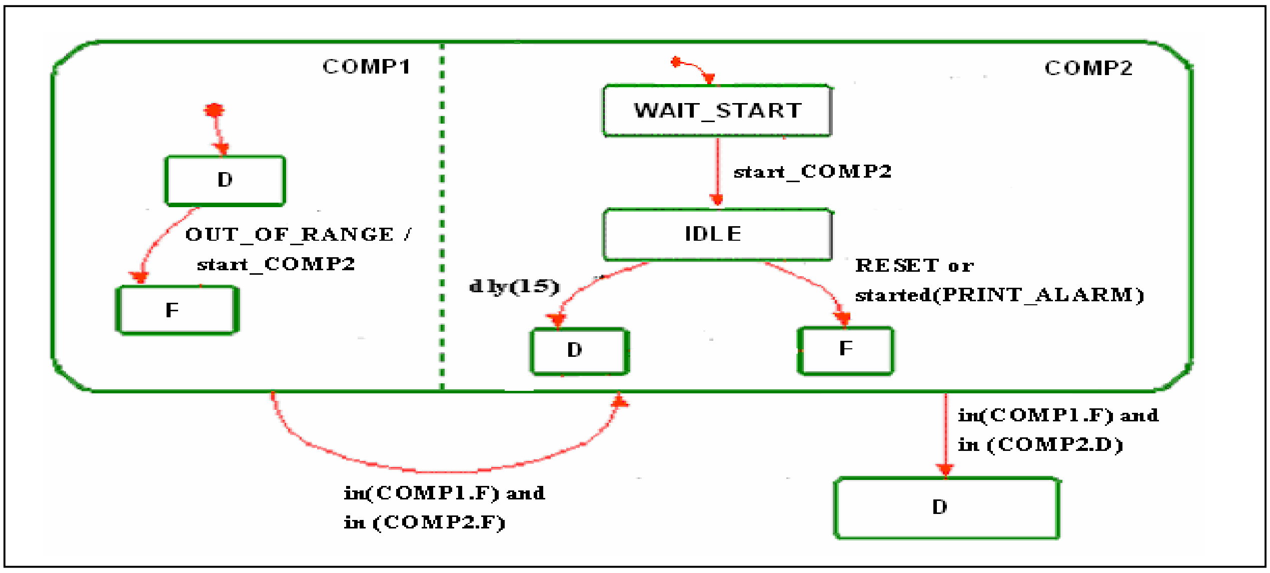
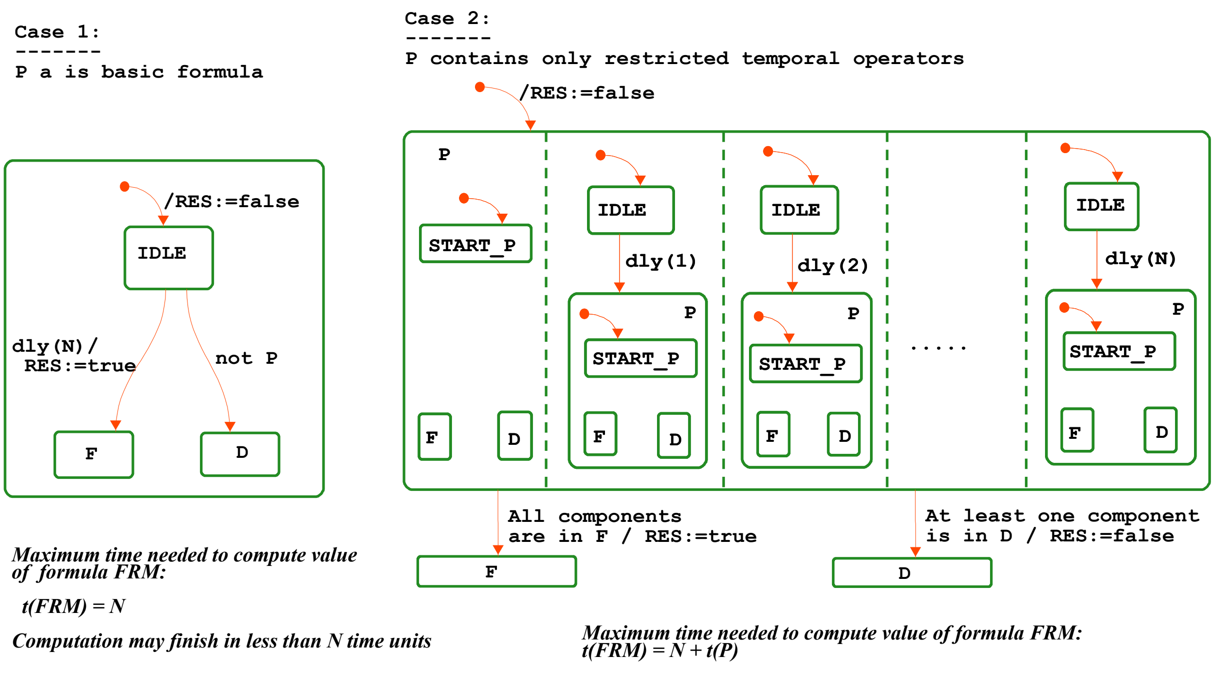


Figure 3. Monitor chart for the assertion
ALWAYS (OUT_OF_RANGE → EVENTUALLY (15) (RESET or started(PRINT_ALARM)))



Case 1:
P a is basic formula
/RES:=false
IDLE
dly(N)/ RES:=true
not P
F
D
Maximum time needed to compute value of formula FRM:
t(FRM) = N
Computation may finish in less than N time units
Case 2:
P contains only restricted temporal operators
/RES:=false
P
START_P
F
D
IDLE
dly(1)
dly(2)
dly(N)
.....
All components are in F / RES:=true
At least one component is in D / RES:=false
Maximum time needed to compute value of formula FRM: t(FRM) = N + t(P)

Figure 4. Translation patterns for formula *ALWAYS (N) P*

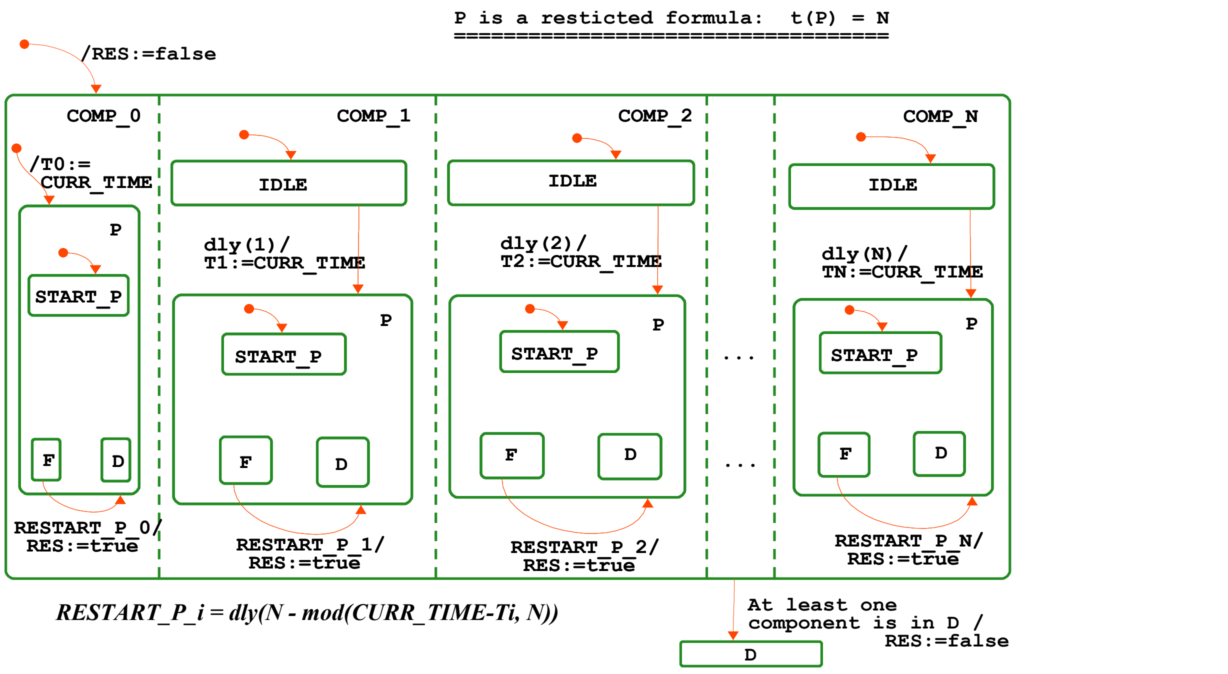


Figure 5. Translation pattern for formula *ALWAYS P*