Locating Features in Distributed Systems

Sharon Simmons, Dennis Edwards, Norman Wilde
Department of Computer Science, University of West Florida
11000 University Parkway, Pensacola, Florida, USA

Abstract

In distributed systems, just as in conventional software, it is often necessary to locate the software components that implement a particular user feature. Several dynamic analysis methods have been proposed to address this feature location problem in conventional software. Most compare traces of execution that exercise different combinations of features.

The feature location problem for distributed systems has complexities beyond those found in sequential systems; namely, concurrent processes and lack of a total ordering of events. This paper introduces a dynamic analysis technique for distributed systems that addresses these complexities. Our methodology defines a component relevance index that can be computed for each software component in the system. Repeated execution of the feature yields more precise indexes for the components. The software components can then be ranked to identify those most relevant to the task at hand.

A small case study is presented to illustrate the formalism and how it might be used in practice.

Keywords: Distributed Systems, Feature Location, Causality, Software Component Location

1 Introduction

Despite all the other changes in Computer Science in the last 30 years, a large proportion of programmer time continues to be spent dealing with existing code. Whether the task is described as "software maintenance", "software evolution", "incremental development" or "component integration", it involves understanding complex code, often written by others, and then either modifying the code or tailoring new code to work with existing code. Such tasks are difficult and error-prone partially because the existing code is often difficult to understand.

Since software changes are often initiated by users, their change requests often refer to "features" of the software. User features rarely involve a single software component. The interactions of several components must be understood to effectively and safely make a change.

Thus while a user of a software system may view it as providing a series of features, to make a change the programmer must view it as a collection of software components. The *feature location problem* involves deducing the mapping between a feature and the key software components involved in its implementation. This problem has received some attention in the Software Engineering literature for sequential programs. Several proposals are based on programming slicing[1], static analysis [2, 3], and dynamic analysis[4, 5, 6, 7, 8, 9].

Our goal is to develop a feature location methodology for distributed systems. A feature in a distributed system is likely to involve not only interactions between different software components, but also interactions of components executing concurrently on different processors. The objective is to provide a mapping between a *feature* and the key *software components* involved in its implementation. For our purposes, a *feature* is any episodically occurring service provided to a user. The user may choose to define the system's features in any way he chooses, provided only that he can identify when each feature occurs and when it does not. Features are most commonly initiated by user input but even a bug may be thought of as a feature if it is observable and identifiable.

The definition of *software component* is likewise flexible. Software components may be classes, individual objects, subroutines, specific lines of code, or even message types. The only restriction is that it must be possible to observe when components are used.

2 System Model and A Component Relevance Index

We model a distributed system as a set, $C = \{c_1, c_2, \ldots, c_q, \ldots, c_Q\}$, of *software components*. As software components execute, they generate events in the N different processes of the system, $P_1 \ldots P_N$. The events on process P_i are identified by the set E_i. All events in $E_i = \{e_i^1, e_i^2, \ldots, e_i^j, \ldots\}$ are totally ordered temporally.

The set E contains all events in the system and is the union of the events from each individual process,i.e., $E = E_1 \cup E_2 \cup \ldots \cup E_N$. The events in E are only partially ordered.

A software component, c_q, is the *source* of an event, e_i^j, if event e_i^j was the direct result of the execution of c_q. The function $\delta()$ maps the event to the source component. That is, $c_q = \delta(e_i^j)$ if c_q is the source of e_i^j.

While each event will map onto a single component, the inverse does not hold. Consider the execution of a single statement component within a loop. Each time the component is executed, it produces a unique event. All the events created by the execution of that statement will map back to the single component.

Figure 1 shows our system model. The left side of the figure shows the code components. These code components are executed giving rise to events in different processes, as shown in the time lines on the right of the figure. The

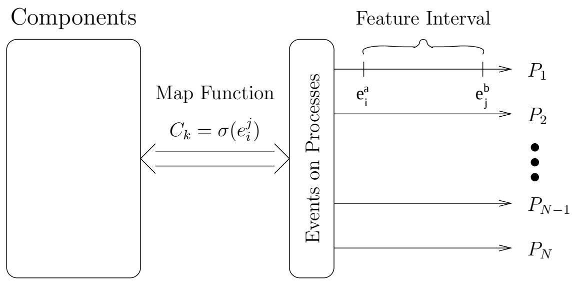

Figure 1: The System Model

mapping between the two is provided by the function $\delta()$.

We define causal relationships among events following the work of Lamport[10]. Event e happens before event e', written $e \rightarrow e'$, if the execution of e can have a causal affect on the execution of e'. The three conditions for $\rightarrow$ are as follows.

1. Events e and e' are executed in the same process and e temporally precedes e'.

2. Event e is the transmission of message m and e' is the receipt of the same message m.

3. Event $e \rightarrow e''$ and $e'' \rightarrow e'$.

Each execution of a feature is associated with an *interval* of events defined by the start event of the feature e_i^a and the end event of the feature e_j^b. The interval consists of the start event, the end event, and all events that both causally follow the start event and causally precede the end event. Note that the start event and the end event may be on different processes. The set I_k for the k'th execution of the feature contains the events in the interval.

$$I_k \quad = \quad \{e : e_i^a \rightarrow e \wedge e \rightarrow e_j^b\} \cup \{e_i^a, e_j^b\} \quad (1)$$

Suppose we have f observations of the feature that give us f intervals in which the feature was active. We use I^* to represent the set of such intervals, $I^* = I_1 \cup I_2 \cup \ldots \cup I_f$.

We define a *component relevance index*, p_c, as the proportion of executions of component

c that occur when the feature is active. The value of p_c ranges from 0.0 to 1.0. If a component is strongly related to a feature then most of its executions will occur when the feature is active and p_c will be close to 1.0. At the other extreme, unrelated components, executed only when the feature is not active, will have a p_c of 0.0. The following weighting function is used to distinguish events that are, or are not, in the combined interval.

$$\omega(e) = \begin{cases} 1 & \text{if } e \in I^* \\ 0 & \text{if } e \notin I^* \end{cases} \quad (2)$$

The weights of different events are combined using the following estimator for p_c:

$$\hat{p}_c = \frac{\sum\limits_{e:c=\delta(e)} \omega(e)}{|\{e : c = \delta(e)\}|} \quad (3)$$

If the data consists of a small number of intervals there can be substantial error of estimation. Suppose, for example, that we begin the execution of the system and take three one-minute intervals, one every fifteen minutes. And, if by chance, there is also a timer triggered event that happens twice an hour. If that timer expires during the first interval, then it will again expire during the last interval. All of the timer events would have occurred in I^* and $\hat{p}_c$ would be 1.0, although there is only a coincidental relationship between the timer event and the feature.

The solution is to collect more data, executing the feature at random but following a consistent operational profile. The errors in $\hat{p}_c$ caused by such accidents should diminish leaving values near 1.0 for strongly related components and values near zero for unrelated components.

3 A Case Study

A small case study serves to illustrate the application of the component relevance index. The *Gunner* program is a simple text-based game developed as a programming exercise in several courses. It simulates a medieval gunner firing a cannon at a castle. The program has two main features: *move the gun* and *take a shot*. The program calculates the trajectory of a cannonball based on user input and draws the trajectory on the screen.

The case study used a distributed version of `Gunner` written using MPI[11, 12]. One process maintained information on the gun and the gunner, another tracked the list of players and their scores, a third computed the trajectory path, and a fourth provided a user interface. The processes were distributed across a Linux cluster. Process interaction was restricted to asynchronous MPI message passing. The different processes collectively amounted to approximately 1900 lines of C++ source code.

We instrumented `Gunner` at all function entries and exits, as well as at all MPI function calls which resulted in 114 software components. Sequence numbers were appended to MPI messages to allow causal relationships of events to be derived from the traces [13]. To simulate the operation of a more complex system, delays were inserted into each function and random noise events were added, as might be caused by background processing.

Two different circumstances are examined in detail. In the first, intervals are approximately delimited since complete instrumentation is either not possible or requires knowledge of the system not currently possessed. The second circumstance has the advantage of precise interval endpoints generated by code instrumentation.

To see the effect of approximate versus precise intervals in the study of `Gunner`, we identified the code in the user interface process where the *move the gun* feature is initiated and completed. Instrumentation generated precise start and stop events. To approximate imprecision in defining the start and stop times, we added a random time between -1 and 1 second to the precise intervals. With precise intervals, the only source of error was the random background noise while, with the generated approximate intervals, incorrect start and stop times

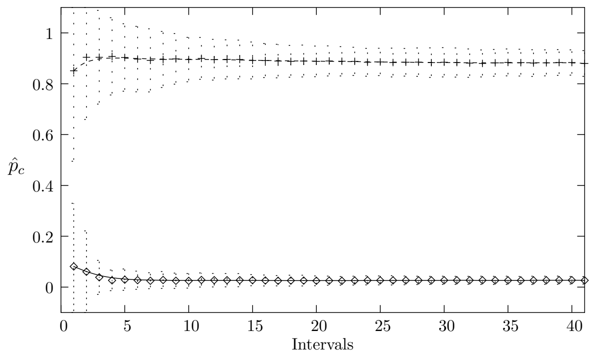

Figure 2: $\hat{p}_c$ values for related and noise

provided an additional source of errors.

The goal of the case study was to investigate properties of the feature location methodology that would be important to a programmer. Specifically, we sought to identify the number of features repetitions needed before $\hat{p}_c$ values converge. Additionally, we wanted to quantify the discrimination $\hat{p}_c$ values made between feature related and unrelated components.

A driver program executed Gunner repeatedly using an operational profile of 10% *move the gun* to 90% *take a shot* operations. To observe convergence, values of $\hat{p}_c$ for the *move the gun* feature were calculated using from one through forty feature repetitions. The sequence was repeated 50 times from which the mean and standard deviation were computed.

Figure 2 shows the convergence of the $\hat{p}_c$ values. The horizontal axis is the number of intervals I_k. The bars show the mean and standard deviation of the $\hat{p}_c$ for that number of intervals.

The top results are for a typical strongly related component where the bottom results are for background noise. As can be seen, $\hat{p}_c$ for this component converged quickly to a value of 0.88. A value of 1.0 is not attained because the simulated user error introduces a small, but significant, probability that an event generated by the component is not in any of the collected intervals. However, 0.88 is among the largest of the $\hat{p}_c$ values found (see the third row in Table 1) so the methodology establishes this component as one that the programmer should investigate. Using precise start and stop times resulted in the software component having a $\hat{p}_c$ of 1.0 in all runs.

The $\hat{p}_c$ is low throughout for a typical noise component, converging to a value of 0.03. The results for precise intervals are similar, since the errors in start and stop times do not effect the probability of seeing a random noise event.

One final way of analyzing the results is to consider the *ranking* of components. We imagine that a programmer repeats the feature several times until the $\hat{p}_c$ values roughly stabilize, and then investigates the top 10% to 15% as important for the feature.

In the `Gunner` study, we chose to examine the top 10% of the 114 components. These eleven rankings stabilized after 10 intervals. Table 1 lists these components showing the mean and standard deviation of $\hat{p}_c$ values for precise and approximate intervals. There is a clear break in the table between the top nine components and the others. The top nine have a $\hat{p}_c$ above 0.89 for precise intervals and above 0.49 for approximate intervals, while the remaining components have values of roughly 0.10 or less.

A check of these components in the `Gunner` code confirms that the top nine are, in fact, heavily used in the *move the gun* feature. The remaining two components are involved in producing the screen display and are shared by both *move the gun* and other features. Note that the $\hat{p}_c$ values for precise intervals differ from the $\hat{p}_c$ for approximate intervals, but the same top components are identified. In both cases, the methodology provides a sharp discrimination between components that are important for the feature and components that are shared or unrelated.

4 Conclusions

This paper has described a method for software component location in distributed systems using dynamic analysis. We have shown that time intervals containing the execution of a feature may be defined based on Lamport's work on causal orderings of events. The component

Function	Event	Precise	Approximate
`validate_gunner`	Entry	1.0000 ± 0.0000	0.9460 ± 0.0699
`validate_gunner`	Exit	1.0000 ± 0.0000	0.8838 ± 0.0981
`moveGun`	Send	1.0000 ± 0.0000	0.8838 ± 0.0981
`gunner_display`	Entry	1.0000 ± 0.0000	0.8838 ± 0.0981
`doEmptyLayoutWithGunner`	Entry	0.8970 ± 0.0278	0.7395 ± 0.1106
`gunner_display`	Exit	1.0000 ± 0.0000	0.5407 ± 0.1633
`moveGun`	Exit	1.0000 ± 0.0000	0.4896 ± 0.1594
`moveGun`	Entry	1.0000 ± 0.0000	0.4980 ± 0.1715
`doEmptyLayoutWithGunner`	Exit	0.8970 ± 0.0278	0.4829 ± 0.1471
`Graphics`	Entry	0.1048 ± 0.0373	0.0906 ± 0.0331
`Graphics`	Exit	0.1048 ± 0.0373	0.0906 ± 0.0331

Table 1: Top 10% of $\hat{p_c}$ components

relevance index may then be computed using traces of intervals in which the feature was active. The component relevance index is subject to statistical error, but can be addressed by repetitive execution of the feature.

While the `Gunner` case study is not necessarily representative of results expected with production distributed software, the component relevance index values did converge rapidly and discriminate sharply. The components identified were important for the understanding of the feature.

We believe that this work is a step toward the development of effective tools to assist programmers in meeting the challenges presented by distributed software systems. Incorporating our methodology into monitoring systems, ranging from laboratory testing to field systems, can provide insight into software component location.

References

[1] Mark Weiser, "Programmers use slices when debugging," *Communications of the ACM*, vol. 25, no. 7, pp. 446–452, July 1982.

[2] Ted Biggerstaff, Bharat Mitbander, and Dallas Webster, "Program understanding and the concept assignment problem," *Communications of the ACM*, vol. 37, no. 5, pp. 72–83, May 1994.

[3] Kumrong Chen and Vaclav Rajlich, "Case study of feature location using dependence graph," in *Proceedings of the 8th International Workshop on Program Comprehension - IWPC 2000*, Los Alamitos, California, June 2000, IEEE Computer Society, pp. 241–249.

[4] Norman Wilde and Michael Scully, "Software reconnaissance: Mapping program features to code," *Journal of Software Maintenance: Research and Practice*, vol. 7, pp. 49–62, January 1995.

[5] J. Deprez and A. Lakhotia, "A formalism to automate mapping from program features to code," in *Proceedings of the 8th International Workshop on Program Comprehension - IWPC 2000*, Los Alamitos, California, June 2000, IEEE Computer Society, pp. 69–78.

[6] W. Eric Wong, Joseph R. Horgan, Swapna S. Gokhale, and Kishor S. Trivedi, "Locating program features using execution slices," in *1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology*, March 1999, p. 194.

[7] Hira Agrawal, James Alberi, Joseph Horgan, J. Jenny Li, Saul London, W. Eric Wong, Sudipto Ghosh, and Norman Wilde, "Mining system tests to aid software maintenance," *IEEE Computer*, vol. 31, no. 7, pp. 64–73, July 1998.

[8] Norman Wilde, Michelle Buckellew, Henry Page, and Vaclav Rajlich, "A case study of feature location in unstructured legacy fortran code," in *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering - CSMR'01*. IEEE Computer Society, March 2001, pp. 68–76.

[9] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon, "Incremental location of combined features for large-scale programs," in *Proceedings of the IEEE International Conference on Software Maintenance*, Montreal, Canada, October 2002, pp. 273–282.

[10] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[11] Message Passing Interface Forum, "MPI: A message-passing interface standard (version 1.1)," Tech. Rep., MPI-Forum, http://www.mpi-forum.org, 1995.

[12] Message Passing Interface Forum, "MPI-2: Extensions to the message-passing interface," Tech. Rep., MPI-Forum, http://www.mpi-forum.org, July 1997.

[13] J. M. Hélary, G. Melideo, and M. Raynal, "Tracking causality in distributed systems: a suite of efficient protocols," in *SIROCCO 2000: The 7th International Colloquium on Structural Information and Communication Complexity*. June 2000, pp. 181–195, Carleton University Press.