Behavior Analysis based Program Understanding to support Software Maintenance

Heinz Pozewaunig and Dominik Rauner-Reithmayer
Institut for Informatics-Systems
Klagenfurt University
Austria
email:{hepo, dominik}@ifi.uni-klu.ac.at

ABSTRACT
One of the most important tasks in software maintenance is to understand the behavior of the system's parts one is dealing with. The common way for a maintainer then is to study the documentation of a system. However, more often than not, this documentation is far away from being up to date, which is due to the system's continuous changes which are not reflected in its documentation. In such a situation a maintainer is lost and her/his only alternative is to dig in the system's source code. This paper deals with the problem of how to infer a (part of a) system's behavior without having to look at its source code directly when the documentation is not trustworthy. For this purpose we analyze the events which are emitted by a system. Such sequences of events contain much information which allow to reason about the behavior of the emitter. We present an approach to infer formal descriptions from event sequences and discuss how these descriptions support the maintenance of a system.

KEY WORDS
Program Understanding, Redocumentation, Software Maintenance, Behavior Sampling

1. Motivation

From whatever perspective you approach it, software maintenance usually poses a challenge to a large variety of aspects for several reasons. One of the most aggravating facets of maintenance is caused by the *evolution* of a system. The term software evolution was first coined by Lehmann [1]. He identified two phenomenons co-occurring with long running systems, which are called

1. *law of continuing change* and
2. *law of increasing complexity*.

Continuing change implies the recurring need to adapt software systems to evolving requirements. A system which is not adaptable diminishes in value over time. Consequently, this leads to the second law, which points out that a system does not only grow in size but also in complexity. Corrective and adaptive maintenance renders an ever lasting activity and they cause impacts leading to the second law of Lehmann. This is due to the effects which result from adapting software systems to changing requirements, thereby increasing the complexity of the system maintained because changes are not reflected adequately in the system's documentation [2].

Systems can be structured according to their static or their dynamic aspects. Maintenance focusing on the static part changes the system's structure (architecture). If such changes are not well documented, the system's decay is inevitable. This is the reason why the adaptations of the static structures are in most cases reflected sufficiently in the systems documentations. The research community is well aware of this fact and there are many approaches aiming at the redetection of static structures, like objects (classes) in legacy systems [3, 4, 5, 6, 7, 8].

However, maintenance activities do not always lead to changes in a system's structure. Many requirements demand small adaptions of the behavior without touching static parts. Although the importance of documenting behavioral changes is well known, this task is usually neglected to a great extent. Thus, the consequence is a *mismatch* between the *actual description* of a system's behavior and its *actual functionality*. Maintenance in such an environment is a extremely difficult task. This problem has been approached from different perspectives in literature [9, 10, 11, 12]. Most of these techniques are based on results of source code analysis and can be applied to analyze behavior of state-less components on the level of procedures and functions. Additionally, a repository containing knowledge in form of behavior patterns (cliches) is necessary to infer a component's behavior. In most cases inferencing is performed by matching chunks obtained from the maintained system with patterns retrieved from the repository. The approaches differ in their extraction techniques and/or their matching algorithms.

Due to their main resource, which is source code, all these approaches for program understanding either produce a large overhead for matching source code patterns or deliver inaccurate results. There are only a few approaches which do not depend on source code; nevertheless, then a full blown knowledge base is needed for a successful application [13, 14].

Even though all of these approaches lead to satisfying results within their domains, they do not take this very important aspect of describing the behavior of *state-bearing* components into account. They operate on a level which is too fine granular to capture the relationships between procedures and functions, resp. methods of one class. Thus, intra-object behavior can not be analyzed by them.

The work presented in [15, 16] aims at the description of the dependencies established between different methods of one object to gain a more holistic insight into its behavior. This dynamic view is provided by state-charts similar to UML-statechart diagrams [17]. To infer this view a very expensive source code analysis assisted by manual guidance is needed, even if the underlying system is rather small. Spending that much effort is not in all cases justifiable.

In this paper we present an approach for program understanding which does not rely on source code analysis. Instead, the starting point of our approach is a knowledge base built upon call traces. These traces are obtained by analyzing simple activity log files, test data, or dynamic traces. From that formal grammars are infered which describe the regularities of event traces. These grammars reflect the main particularities of the behavior of state-bearing objects. The main assumption of our approach is that traces can be produced (or isolated) for those parts of a legacy system where the maintenance focus is placed. In the remainder of this paper we refer to the maintained parts as components, e.g. objects, modules, clusters of them.

The grammars, as well as statechart diagrams derived from that grammars describe all legal event sequences which were observed in the analyzed data. These event sequences comply with meaningful behavior of executed objects. Thus, the grammars resp. statechart diagrams reflect the dynamics of the objects in observation.

In the next section, the technique of how grammars are derived from event traces is discussed in detail. In addition the necessary qualities of event sequences are mentioned. How the results of the inference process are interpreted to obtain a meaningful description for objects' behavior is presented in section 3.2. The benefits of our approach are shown in section 3., where the results are applied to a maintenance problem. We conclude this paper with a discussion about further work to be done and we give a short subsumption.

2. Inferring formal grammars from Event Traces

2.1 Event Trace

An event trace is a sequence of procedure- or method-calls of one component. A component is either an object or any coherent structure (e.g. module) comprising one state space. In this work we neglect concrete values which are transformed by a procedure - or method call to get an abstract view of the trace data. Traces can be obtained in two different ways: (1) by analyzing the call graph in the source code to build a static call trace, or (2) by observing the actual occurring calls during run time forming a dynamic trace. Static traces contain more information about the potential behavior of a component, which is due to the inclusion of all paths given in the source code. Whereas for maintenance it is not always necessary to have complete knowledge available, since only those paths are interesting which demonstrate the behavior to be maintained. Furthermore, the analysis of source code would be too costly, if only a particular behavior needs to be analyzed, which can be produced by the system anyway.

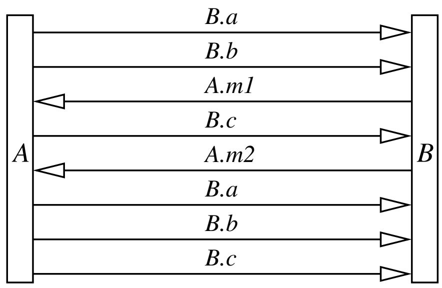

Figure 1. A communication sequence producing traces

An event trace is gathered by observing calls to a particular component. In figure 1 an UML sequence diagram demonstrates the communication between component A and component B. This scenario contains two different traces which are an example for the behavior of A and B. The trace of calls of component A's methods can be identified as $A.m_1\ A.m_2$; that of B's methods is $B.a\ B.b\ B.c\ B.a\ B.b\ B.c$. If traces are selected intelligently, the examples describe a significant part of behavior. Obviously, the complete characteristics of a component can hardly be captured by such traces. Therefore, we have to abstract from them.

In UML, the dynamics of components is described by means of statechart diagrams which are extended finite state automatons (FSA) [17]. Hence, event sequences are considered as words of a formal language, specified by a FSA. Our aim is to infer the automaton from words, resp. event traces.

2.2 Inference process

The process of inferencing grammars from words belongs to the problem field of sequence learning [18]. For maintenance one important fact must be considered: the knowledge about components exists as *positive* examples solely. Positive examples are words produced by the grammar, whereas negative examples do not belong to the language at all. Negative examples turn out to be very valuable, since they restrict the search space tremendously. Therefore, the problem class is further refined and named in literature as *positive grammar inference problem*.

In 1978 Mark Gold [19] proved, that in general, the identification of a canonical automata[1], on the basis of given positive and negative examples, is NP-hard. However, if only positive examples were available, in general such an identification would be impossible. In the mid 80ies, therefore, research concentrated on heuristics in order to build automata for a specific context.

We adapted the SEQUITUR algorithm [20] to our needs. SEQUITUR generates a context-free grammar [21] from a given word. In fact, although the grammar belongs into this class of complexity, the generated languages are regular, since each infered language is finite and it can produce one word only. SEQUITUR analyzes a sequence and substitutes each repetition by a rule. The aim of the algorithm is twofold: (1) It generates a shorter representation of the input, thus encoding it as grammar. (2) As a by-product a structural explanation of this sequence is provided.

A context free grammar G is a 4-tuple $G = (T, N, R, S)$ containing a set of terminals T, a set of non-terminals N, a set of rules R of the form $A \rightarrow \alpha$, where A is a non-terminal symbol and α denotes a string of terminal and non-terminal symbols, and, finally, the start symbol S. For a rule in that form A is called the rule's name as well. Consider the sequence of component B (figure 1), which contains a repetition of the subsequence $a\ b$. SEQUITUR replaces them with a nonterminal and introduces a new rule with that symbol. The grammars produced share the following two properties:

Digram uniqueness No pair of adjacent symbols exists more than once in the grammar. Adjacent symbols are substituted by grammar rules.

Rule utility Every rule is used more than once. If a rule is used only once in the grammar, it is deleted and the place where it was utilized is substituted by its right hand side.

The following grammar is infered from the example sequence (nonterminals start with the meta symbol $, the grammar's start symbol is named an **S**):

S $\rightarrow$ $A $A
$A $\rightarrow$ $a\ b\ c$

2.3 Extensions to SEQUITUR

SEQUITUR is adequate for describing single words. However, in a maintenance context, many words (event traces) of one component are given, each of which represents one specific detail of behavior. Only when put together, a valid description is producible. Given are the following two event traces of one component, $a\ b, c$ and $b\ c\ a$. The SEQUITUR algorithm analyzes this examples independently and therefore the two grammars constructed do not find the evident behavior pattern $b\ c$. To overcome this weakness, SEQUITUR was extended to enable the analysis of many different sequences, which are assumed to be produced by one component. This extension is called MSEQ (multiple SEQUITUR) [22]. MSEQ detects repeating patterns spread between different words.

Due to the primary goal of SEQUITUR, which is the grammar encoding of one word, only the word analyzed can be derived from the grammar. In particular, iterations of contiguously appearing tokens, which may refer to loops in the source code, are not explicitly revealed by SEQUITUR. This is not sufficient in program understanding and thus, a more general and abstract description of a component's behavior is needed. MSEQ provides with the possibility to introduce domain knowledge in form of an iteration *heuristic*. An iteration heuristic is the minimal number of sequentially occurring patterns in a word. If at least that many patterns occur consecutively, it is assumed that these patterns were produced by a loop. For instance, the event trace $a\ b\ b\ b\ b\ c$ hints to a grammar G with the rules $\mathbf{S} \rightarrow a\ B\ c$, $B \rightarrow b \mid b\ B$, where an infinite sequence of the terminal b may be produced. Obviously, this pattern does not need to be generated by a loop in the component emitting the sequence. How many sequential occurring patterns really hint to a loop depends on domain knowledge. Therefore, the iteration detection mechanism of MSEQ can be controlled by the `minimalIt`-parameter, which specifies the minimum number of iterations defining a loop. In the example, above the values 2, 3, or 4 would have led to the same result. The correct adjustment of this parameter depends on domain expertise and must be provided by the maintainer.

SEQUITUR bases its pattern recognition on the detection of repeated digrams (patterns of length 2). However, there may be situations, where the sequences demonstrate a certain pattern length larger than 2. In such cases it is better to base the grammars on that characteristic length. Thus, MSEQ additionally offers the possibility to detect patterns (n-grams) of a length n specified by the maintainer which leads to more domain-adequate rule structures.

3. Maintenance support

3.1 Interpreting grammars for program understanding

MSEQ produces formal grammars representing characteristic parts of a component's behavior. Since they abstract from concrete values which are provided with the calls, MSEQ provides with a more general description, which can hardly be recognized directly from the traces. In this way, grammars help to understand behavior. Due to the fact that many grammars correctly describe the same words, this degree of freedom is used to raise the level of understanding. The maintainer may adapt MSEQ by tuning the parameters

[1] An infered automata is canonical, if it is equivalent to the automaton producing the word.

in order to generate results which reflect the particular domain better.

One possibility is to "play" with parameters to get a notion of the boundaries important in the domain. If the domain contains transactions of a certain length, the results obtained by a default pattern length may not reveal this characteristics. By experimenting with the value n for the n-grams, various grammars describe the underlying words. The responsibility which grammar to chose as the best description for that domain is passed onto the maintainer. For detecting iterations, a similar approach is chosen. A meaningful variation of the `minimalIt`-parameter helps to produce grammars with different levels of quality which generally aims at their understandability.

As a matter of fact, formal grammars are not that easy to comprehend for untrained maintainers. This is especially true if iterations are represented as recursions. Thus, MSEQ can be forced to change the output format of the grammars to EBNF[2]. The recursion of the start rule of the grammar presented in section 2.3 thus becomes $\mathbf{S} \rightarrow a\ (b)^+\ c$, which is indeed easier to comprehend[3].

A further improvement to understandability is due to the property of the grammars generated by MSEQ whose languages are always regular. Regular languages are equal to FSAs [21], which are more understandable in general due to their graphical representation. Thus, these grammars can be transformed to FSAs without losing any information. But states in a FSA are not directly comparable with those of an UML statechart diagram. Anyhow, since they are depicted as graphs, their expressiveness is rated higher than the one of grammars. Algorithms which transform grammars to deterministic as well as indeterministic FSAs may be found in [21].

3.2 Interpreting grammars for maintenance

How can grammars describing partially the behavior of components help to maintain software? When considering the different kinds of maintainance activities, grammars help in the following ways:

In *corrective maintenance* one of the main tasks is to locate the spot where an error occurs. Grammars are able to demonstrate structures of behavior. On a high level they can indicate wrong orders of events, incorrect placed recursions (iterations), or missing iterations. Any of these indicators may show wrong protocols of the component's utilization and, thus, enables the maintainer to locate the corresponding parts in the source code by identifying the functional parts emitting wrong patterns. The localization is supported by two sources of information, which are (1) the event-emitting component and (2) the path within the component's control flow graph leading to error prone places. E.g. see our previously presented example (page 2) which led to the grammar presented on page 3. Given is the following snippet of source code of component A, which may produce sequences of the form $((a\ b\ c)|(a\ b\ d))^+$.

```
WHILE conditon1 LOOP
  call B.a;
  call B.b;
  IF condition2 THEN
    call B.c;
  ELSE call B.d;
  END IF;
END LOOP;
```

The grammar infered does not contain a terminal d. Due to the maintainer's domain knowledge, this is considered a fault since d must show up in the grammar. Thus, the maintainer is hinted to look at `condition2` controlling the path to d, and to look at those variables which are refered to in the condition.

Adaptive maintenance enhances the functionality of a system by changing it in accordance with new requirements. Grammars generated by MSEQ support this activity by guiding the maintainer to those places where changes should happen. Consider a required adaption of the behavior to the example given above, where a word $a\ b\ e$ must be producible. Therefore, a further alternative in the `IF`-construct, `B.e`, should be available when a particular condition holds. Similarly to the path location, during corrective maintenance the grammar directly helps to locate the corresponding spot in the source code.

4. Further Work

Grammars describe the behavior of components. We provided hints for locating spots to be maintained in source code on the basis of grammars and words. These hints could be further improved, by connecting the forward engineering process for developing components to the program understanding activity and maintenance stronger. In that direction some work has already been done [2], where formal specifications for establishing forward and backward traces are utilized. We want to exploit these techniques for program understanding as well.

MSEQ detects any repetitions in event traces. If characteristic patterns of the respective domain were known in advance, the inference process as well as the resulting grammar would benefit tremendously from the use of them. Hence, a next step to improve MSEQ would be to develop the algorithm into that direction.

5. Conclusion

We presented an approach to automatically generate descriptions of behavior from dynamic event traces which are obtained from a running system. After isolating the parts of a system to be maintained, such generated descriptions (formal grammars, finite state automatons) help to understand the behavior of the isolated part. This type of descriptions is a common way to specify the behavior of software

[2] Extended Backus Naur Form

[3] In EBNF an x^+ indicates that x is at least iterated once.

non-ambiguously. Hence, a maintainer does not have to be especially trained to interpret these abstractions. Thus, without touching the source code, a maintainer is able to reason about a system's behavior and, consequently, is supported in its maintenance.

References

[1] Meir M. Lehmann. Programs, life cycles and laws of softwware evolution. In *Proceedings of the IEEE*, volume 68, pages 1060–1076, September 1980.

[2] Helfried Pirker. *Specification Based Software Maintenance (a Motivation for Service Channels)*. PhD thesis, Klagenfurt University, Austria, September 2001.

[3] S. Liu and N. Wilde. Identifying Objects in a Conventional Procedural Language: an Example of Data Design Recovery. In *Proceedings of the International Conference on Software Maintenance*, pages 266–271. IEEE, IEEE Computer Society Press, 1990.

[4] H. P. Haughton and K. Lano. Objects Revisited. In *Proceedings of the International Conference on Software Maintenance*, pages 152–161, Sorrento, Italy, October 1991.

[5] Harry M. Sneed. Migration of Procedural Oriented COBOL Programs in an Object-Oriented Architecture. In *Proceedings of the International Conference on Software Maintenance*, pages 105–112, Orlando, Florida, November 1992.

[6] Harald C. Gall, Rene Klösch, and Roland T. Mittermeir. Long Term Information Systems Evolution via Object-Oriented Re-Architecturing. In *Proceedings of the 6^{th} European Software Engineering Conference, ESEC '95*, 1995.

[7] G. Canfora, A. Cimitile, and G. A. Di Lucca. Recovering a Conceptual Data Model from COBOL Code. In *Proceedings of the 8^{th} International Conference on Software Engineering and Knowledge Engineering – SEKE 96*, pages 442–449, Lake Tahoe (Nevada), June 1996.

[8] Christian Lindig and Gregor Sneltig. Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis. In *Proceedings of the 19^{th} International Conference on Software Engineering ICSE'97*, pages 349–359, Boston, USA, May 1997.

[9] Charles Rich and Linda M. Wills. Recognizing a Program's Design: A Graph-Parsing Approach. *IEEE Software*, pages 82–89, January 1990.

[10] Santanu Paul and Atul Prakash. A Framework for Source Code Search Using Program Patterns. *IEEE Transactions on Software Engineering*, 20(6):463–474, June 1997.

[11] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. Program Understanding and the Concept Assignment Problem. *Communications of the ACM*, 37(5):72–82, May 1994.

[12] K. A. Kontogiannis, R. Demori, M. Galler, and M. Bernstein. Pattern Matching for Clone and Concept Detection. *Automated Software Engineering*, 3:77–108, 1996.

[13] Elizabeth Burd and Malcolm Munro. A method for the identification of reusable units through the reengineering of legacy code. *The Journal of Systems and Software*, 44(2):121–134, December 1998.

[14] Heinz Pozewaunig and Dominik Rauner-Reithmayer. Support of Semantics Recovery during Code Scavenging using Repository Classification. In *Symposium on Software Reusability – SSR'99*, pages 65–72, Los Angeles, CA, May 1999. ACM.

[15] Dominik Rauner-Reithmayer and Roland T. Mittermeir. Behavior Abstraction to support Reverse Engineering. In *International Conference on Software Engineering and Knowledge Engineering SEKE'98*, San Francisco, USA, June 1998.

[16] Dominik Rauner-Reithmayer and Roland T. Mittermeir. Zustand-Ereignis Diagramme zur Untersttzung des Software Reverse Engineering. In *2^{nd} Workshop Software-Reengineering*, Bad Honnef, Germany, May 2000.

[17] Unified Modeling Language (UML) 1.3 specification. Object Management Group, Needham, MA, USA, `http://www.omg.org/technology/documents/index.htm`, current September, 2001, March 2000.

[18] Ron Sun and C.L̃ee Giles, editors. *Sequence Learning – Paradigms, Algorithms, and Applications*, volume 1828 of *Lecture Notes in Computer Science*. Springer Verlag, 2001.

[19] E. Mark Gold. Complexity of Automatic Identification from given Data. *Information and Control*, 10:302–320, 1978.

[20] Craig G. Nevill-Manning and Ian H. Witten. Detecting Sequential Structure. In *Proceedings of the Workshop on Programming by Demonstration, ML'95*, July 1995.

[21] John E. Hopcroft and Jeffrey D. Ullman. *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*. Addison-Wesley, 1990.

[22] Heinz Pozewaunig. *Mining Component Behavior to Support Reuse*. PhD thesis, University Klagenfurt, Austria, October 2001.