Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 94 (2004) 59–69
www.elsevier.com/locate/entcs

A Metamodel for Dynamic Information Generated from Object-Oriented Systems[1]

Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge[2]

School of Information Technology and Engineering, University of Ottawa
800 King Edward Ave., Ottawa, Ontario, Canada

Abstract

Fully understanding object-oriented systems is difficult if one is limited to merely performing static analysis of the source code: Techniques based on dynamic analysis are needed. Several tools exist that implement techniques for analyzing traces of object interactions; however, due to lack of a common exchange format these tools do not interoperate. In this paper, we propose a graph-based format, called Compact Trace Format (CTF), for exchanging traces of object interactions. The design of CTF takes into account the size explosion problem that makes run-time information hard to manipulate. To achieve this, we start by presenting techniques for reducing the size of traces without affecting their content. Then, we show how the concept of transforming the tree representation of a trace into an acyclic compact graph can be used to define the core of CTF schema.

Keywords: Metamodelling, Dynamic Analysis, Traces, Exchange Formats, Compact Trace Format, DMM, Reverse Engineering

1 Introduction

Most of the exchange formats that exist today focus on representing the static relationships between system artefacts [1, 6, 10]. However, the emergence of object-oriented systems has attracted increased attention to dynamic analysis techniques. This is because polymorphism, inheritance and dynamic binding make it particularly difficult to fully understand object-oriented systems by

[1] This research is sponsored by NSERC, NCIT and QNX Software Systems.
[2] Email: {ahamou,tcl}@site.uottawa.ca

doi:10.1016/j.entcs.2004.01.004

merely performing static analysis. In this paper we will focus on the representation of object-oriented traces. However, a subset of our model can be used for procedural traces.

Dynamic analysis of object-oriented systems is typically done by analysing traces of object interactions. Objects interact by sending messages, which result in message invocations. To reproduce a good representation of the execution of an object-oriented system, one needs to collect at least the events related to method entry and exit, as well as object construction and destruction [4]. These can be shown on a UML sequence diagram or, more compactly, using a tree structure. The advantage of the latter representation is that it enables the use of the various algorithms that operate on trees to process the traces.

There are several tools that operate on traces of object interactions to help software engineers in their daily maintenance activities [2, 3, 12, 13, 15, 16]. However these tools have different formats for representing traces, which hinders interoperability. In order to take better advantage of these tools we need to define a common format for exchanging traces of object interactions.

An exchange format consists of two main components: a schema that represents the entities to exchange and their interconnections and the syntactic form of the file that will contain the information to exchange. This paper introduces a schema for representing traces of object interactions called the Compact Trace Format (CTF). The syntactic form can be represented using existing formats such as GXL (Graph eXchange Language) [11] or TA (Tuple Attribute Language) [10] as we will explain in Section 3.

Traces are usually very large and difficult to manipulate. A good exchange format should therefore address scalability problems in an efficient way. To achieve this, we use two techniques. The first technique is based on investigating ways to reduce the size of a trace without affecting its content. For example, repetitions of a method invocation that are due to loops can be removed and replaced with the number of repetitions without affecting the content of the trace. This technique can result in an important reduction as we showed in our previous experiments described in [8].

The second technique takes advantage of the fact that, as with any trace of procedure calls, traces of object interactions form tree structures. Furthermore, any rooted labelled tree can be transformed into a directed acyclic graph by representing identical subtrees only once [5, 7].

The rest of this paper is organized as follows; the next section focuses on the compression techniques. In Section 3, we present the exchange format CTF. In Section 4, we present the application of CTF through an example.

2 Compressing Traces

In this section, we will discuss how traces can be represented more compactly. All the techniques discussed here can be applied in real-time while a trace is being generated, or else by post-processing a trace after it has been generated.

The processes we use are a form of data compression, i.e. reducing the space required for data, while preserving information content. But we cannot use traditional data compression approaches that render the data unintelligible, because we are not interested in *temporarily* shrinking traces for storage and transmission. Rather, we want to enable applications to operate directly on permanently compressed traces. Realistic traces can be so big that applications cannot efficiently process the data in the uncompressed form.

2.1 Compression by Transformation

A key compression approach we will use is based on the fact that any rooted ordered labelled tree can be transformed into its compact form by representing the occurrences of the same subtrees only once [5, 7]. The result of this transformation is a directed ordered acyclic graph. Figure 1b. shows a directed ordered acyclic graph that corresponds exactly to the trace of Figure 1a (to avoid cluttering, the information that describes a given invocation, namely, class, object and method names are represented with one letter). Note that the crossing line allows maintaining the order of calls. For example, the call to the second occurrence of the tree rooted at B comes after the call to E.

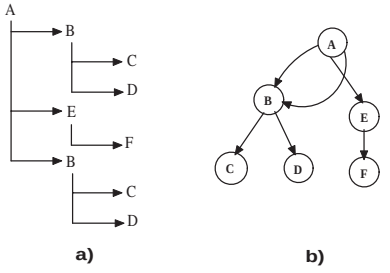

Figure 1. Transforming a tree into a directed acyclic graph

The number of nodes of the acyclic graph is always smaller or equal to the number of nodes of the trace (its tree representation). The number of edges is also reduced. This technique was first introduced by Downy, Sethi and Tarjan [5] to facilitate the construction of tools to explore trees efficiently. An efficient and easy algorithm that does the transformation is presented by Flajolet et al. [7].

We use the term *comprehension unit* to describe a distinct subtree of the trace. From the perspective of comprehension, a software engineer needs to understand this sequence only once and reuse this knowledge whenever it occurs. We refer to the method that triggers a given comprehension unit as: *a comprehension unit initiator*. One can easily notice that the number of nodes of the graph is also the number of comprehension units of the trace since repeated subtrees are represented only once. The trace of Figure 1a. contains 9 calls but only 6 comprehension units as illustrated on the graph of Figure 1b. This is due to the fact that the subtree rooted at B is repeated twice.

2.2 Removing repetitions

Repetitions of method calls in a trace are very common. One reason behind this is the presence of loops and recursion in the source code. Removing such repetitions can result in further compression of the trace. Repetitions of single comprehension units due to loops or recursion can be removed and the number of repetitions can be saved in order to reconstruct the original trace as illustrated in Figure 2.

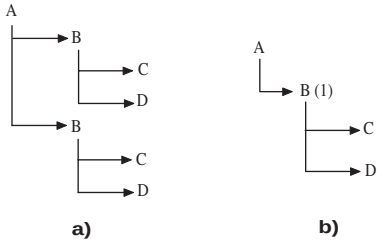

Figure 2. Removing repetitions of single comprehension units

However, repetitions of multiple comprehension units require adding a virtual call that we call a *control node* in order to maintain the hierarchical structure of the trace. Figure 3b. shows a control node called SEQ that is added to the trace of Figure 3a. The number of repetitions is also indicated.

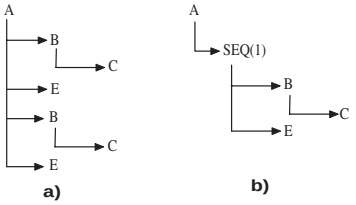

Figure 3. Removing repetitions of multiple comprehension units

Non-contiguous repetitions of comprehension units seem to be the repetitions that have attracted the most attention from the research community.

The common hypothesis is that they can be used to bridge the gap between low-level components and high-level domain concepts. They may appear in the trace due to way the system is used during the generation of the trace. For example, a software engineer may decide to open the same dialog box twice but at different times. This will cause the corresponding implementation to occur in a non-contiguous way in the trace indicating that the same thing had happened. These repetitions are also referred to as patterns that we call *trace patterns*. A trace pattern is shown on the directed acyclic graph as a node with more than one parent as illustrated in Figure 1b.

3 Compact Trace Format (CTF)

In this section, we present the compact trace format (CTF), which is a format for exchanging traces of object interactions. We use a UML class diagram to represent the information represented by CTF. Although, this format can only represent interactions between objects in terms of method calls, we believe that it can be easily extended to represent other kinds of relationships such as interactions among clusters of objects.

3.1 Schema

Figure 4 shows a UML class diagram that describes the schema of CTF. The class Scenario is used to describe the usage scenario from which the execution trace is derived. We allow a scenario to be represented by many traces to support natural situations where many traces of the same scenario are gathered to detect anomalies caused by non-determinism.

The class Trace is an abstract superclass that describes common information that different types of traces may have. This class allows extending the model to consider other type of traces such as traces of inter-process communication. Although, we focus on object-oriented systems, we added the RoutineCallTrace class to represent traces of routine calls of procedural systems. By routine, we mean any function that is not a method of a class. In addition to that, some programming languages such as C++ use routines as well as methods. To deal with this problem, we decided to represent traces of method calls as a subclass of RoutineCallTrace. By doing so, an analyst can create traces of routine calls only, traces of method calls only or traces of routine and method calls such as C++ execution traces. In addition to that, this allows keeping the design simple and understandable.

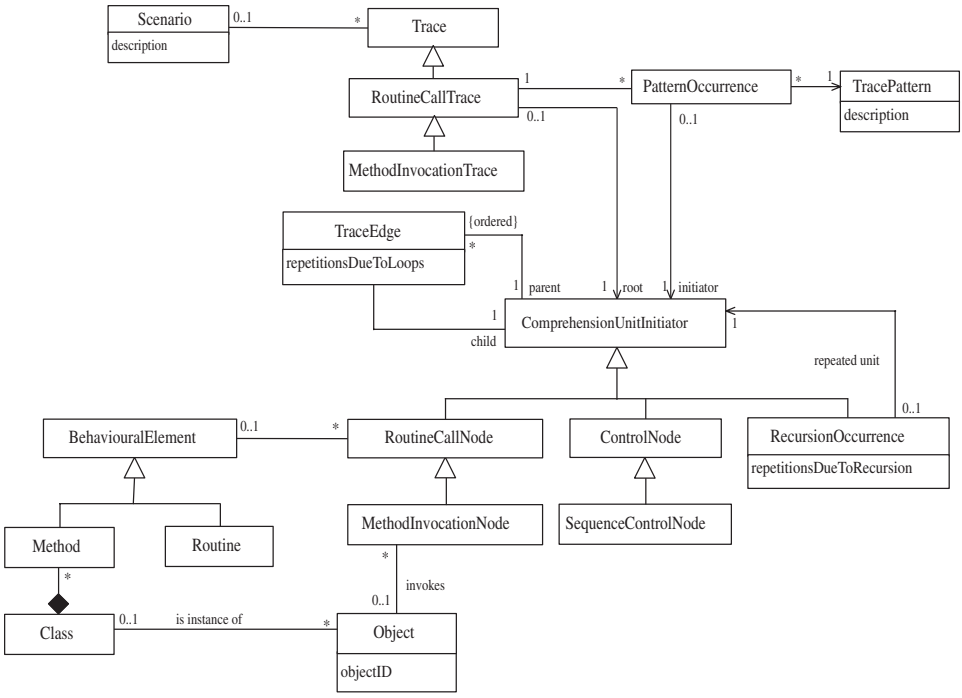

Figure 4. CTF Schema

Since subtrees are represented only once in the graph, a root of any subtree constitutes a comprehension unit initiator, represented using the class ComprehensionUnitInitiator. These are the nodes in the graph; each can have many child nodes and many parent nodes as illustrated on the diagram using the parent and child roles.

Edges (i.e. invocations) are represented using the class TraceEdge. An edge is labelled with the number of repetitions due to loops if there are any.

A node (ComprehensionUnitInitiator) can either be a routine call node, a method node or a control node. Control nodes represent extra information that is needed to manipulate the trace such as the one presented in Section 2.2. The subclass SequenceControlNode represents the attribute SEQ that is used to remove contiguous repetitions of multiple comprehension units. One may also think of several other useful control nodes and extend the ControlNode class to represent them.

Although the schema presented in this paper focuses on describing run-time information, some of its components need to refer to static components of the system. For this purpose, the classes Routine, Method and Class are added and associated to RoutineCallNode to describe references to actual objects representing each class, method, etc. Other existing schema that models the

static components of the system can be used as well. In our case, we chose DMM [9], which is a model for representing software artefacts for various reverse engineering applications.

The class Object and the association that links it to MethodInvocationNode allow having traces that describe invocations at the object level. An object is identified by an object ID.

A recursive comprehension unit is represented with the subclass RecursionOccurrence which in turn refers to the recursively repeated unit. RecursionOccurrence contains the number of repetitions due to recursion.

Trace patterns are represented using the class TracePattern. This class contains one attribute that can be used to assign a high-level description to the trace pattern. The same trace pattern can occur in more than one trace. Indeed, a pattern that occurs in several traces might be more relevant than another pattern that appears in one or two traces only. Relevance, here, is defined with respect to how close the pattern is to a design concept. Finally, the class PatternOccurrence is used to depict in which trace the pattern occurs.

3.2 Syntactic Form

We support the idea that a schema should be defined independently from any syntactic form of the file that represents the data. GXL [11] is one candidate for the syntactic carrier for CTF. A GXL file consists of XML elements for describing nodes, edges, attributes, etc. It was designed to supersede a number of pre-existing graph formats such as GraX [6], TA [10], and RSF [14]. GXL has been widely adopted as a standard exchange format for various types of graphs by both industry and academia.

However, a GXL representation of CTF would tend to be much larger than necessary due to the use of XML tags and the explicit need to express the data as GXL nodes and edges. The compactness benefits of CTF would therefore be partially cancelled out. Whereas the wordiness of GXL would not be a problem when expressing moderately sized graphs in other domains, the sheer hugeness of traces suggests an alternative might be appropriate.

One reasonable alternative syntactic form is TA [10], which would minimize the space required by a CTF trace. Future work should focus on determining which syntactic form suits best information represented by the CTF schema. We also intend to conduct empirical studies to estimate the gain reached by the compression techniques presented previously.

4 Application

In this section, we illustrate the use of CTF schema through an example. First, we explain the different techniques for extracting traces of object interactions. Then we introduce an imaginary trace and model it using CTF. The result is an instance diagram of the CTF schema.

To reproduce the execution of an object-oriented system, one needs to collect at least the events related to object construction and destruction, as well as method entry and exit [4]. Additional information can be collected as well. For example, if the system is a multi-threaded system, then events related to thread execution need to be collected. There are typically three techniques for generating traces of method calls. The first technique is based on instrumenting the source code, which consists of inserting probes (e.g. a print statement) at different locations in the source code. The second technique consists of instrumenting the execution environment in which the system runs. For example, the Java Virtual Machine or bytecode can be instrumented to generate events of interest. The advantage of this technique is that it does not require the modification of the source code. The last technique consists of running the system under the control of a debugger. In this case, breakpoints are set at locations of interest. This technique has the advantage of not modifying the source code and the environment; however, it can slow down considerably the execution of the system.

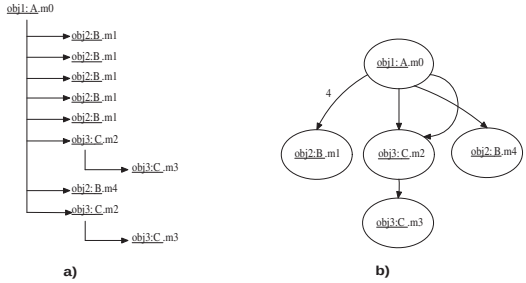

Figure 5. a) An example of a trace as a tree. b) The directed ordered acyclic graph corresponding to the trace a)

To illustrate the use of CTF, let us consider an example: Suppose that the result of exercising a feature of a particular system generates the trace shown in Figure 5a. The compact graph of this trace is shown in Figure 5b.

The trace involves three classes namely A, B and C and three objects, which are obj1, obj2 and obj3. There are five methods that have been invoked: m0, m1, m2, m3 and m4. We notice that the call generated to obj2:B.m1 is repeated five times in the trace, probably due the existence of a loop in the

source code. We also notice that this trace contains a pattern which consists of the sequence of calls generated by obj3:C.m2 as depicted clearly on the directed ordered acyclic graph of Figure 5b. It is interesting to notice that the transformation of the trace into a graph and the removal of contiguous repetitions results in a considerable decrease of the size of the trace. Indeed, the tree contains eight nodes and seven edges whereas its graph representation contains five nodes and five edges.

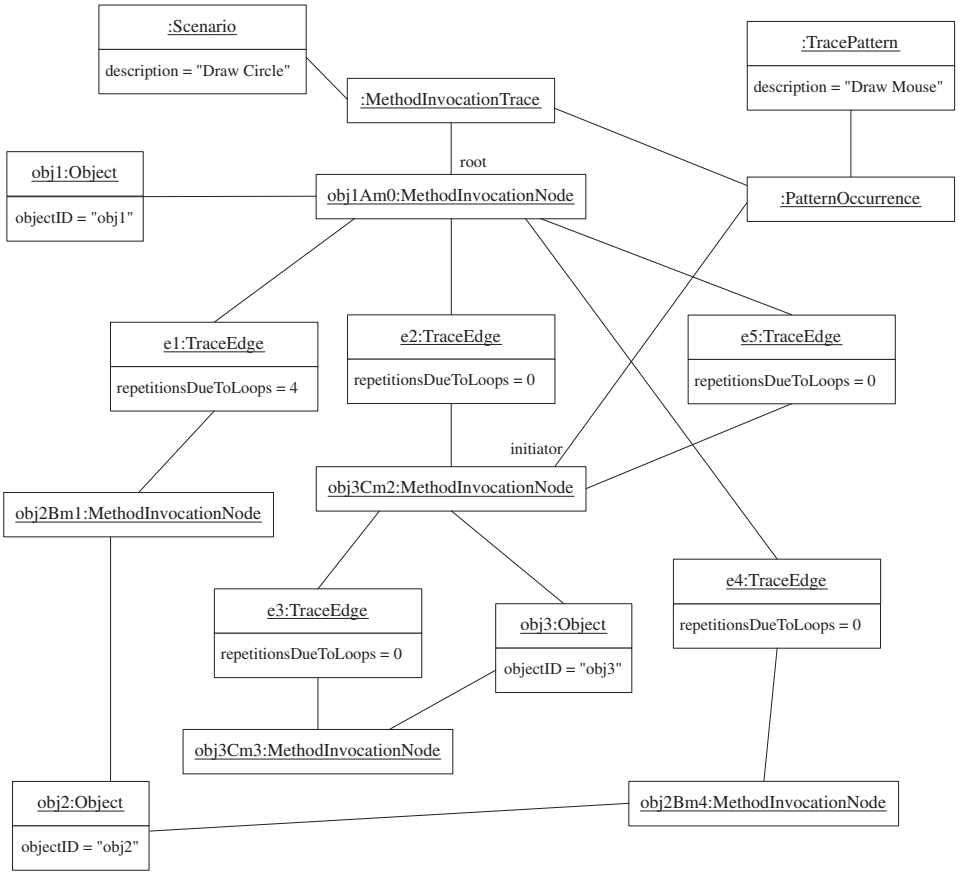

Figure 6. An example instance diagram using the CTF schema

An instance diagram of the above trace using the CTF schema is shown in Figure 6. This diagram omits instances of the static model classes (e.g. Class, Method...) to avoid clutter. We imagine that the overall scenario is called "Draw Circle" (as in a drawing program); this scenario is represented with the object of class Scenario and the trace is depicted by the object of class MethodInvocationTrace. The nodes are represented with the objects obj1Am0, obj2Bm1, obj3Cm2, obj3Cm3 and obj2Bm4. Each node will need

to refer to instances of the static model. Edges are represented using instances of the class TraceEdge. There are five edges.

The node obj3Cm2) has two incoming edges. The software engineerin using the ool can therefore mark this as a pattern, shown here as an instance of PatternOccurrence. The user has indicated, by the 'description' attribute, that this pattern is concerned with the "Drag Mouse" operation.; it The instance of PatternOccurrence shows which particular trace the pattern occurs in, and which nodes are the initiators (one node in this case).

5 Conclusion and Future Directions

A common exchange format is important for allowing different tools to work together. Lack of such a format for exchanging traces of object interactions hinders the interoperability among tools that manipulate these traces.

This paper introduces CTF (Compact Trace Format). We describe the CTF schema using a UML class diagram. To deal with the size explosion problem of execution traces, we transformed the tree representation of the traces into their corresponding directed acyclic graphs, where repeated subtrees are represented only once. We also removed repetitions for a better compression.

Future work should focus on three aspects: The first aspect consists of experimenting with numerous execution traces of many systems to test the efficiency of CTF. By efficiency, we mean the ability of the format to convey information from large system executions as compactly as possible. The second aspect is concerned with the adoption of CTF by tool builders. In order to move towards this, we need to arrange to have several tool builders use CTF, or a descendent, as their standard format. Finally, we need to extend CTF to handle other types of execution traces such as traces of inter-thread communication.

References

[1] I. T. Bowman, M. W. Godfrey, and R. C. Holt, "Connecting Architecture Reconstruction Frameworks", *Proc. 1st Int. Symp. on Constructing Software Engineering Tools (CoSET'99)*, Los Angeles, CA, 1999, 43-54.

[2] W. De Pauw, R. Helm, D. Kimelman, J. Vlissides, "Visualizing the Behaviour of Object-Oriented Systems", *Proc. 8th Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA),* 1993, 326-337.

[3] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, J. Yang, "Visualizing the Execution of Java Programs", *Proc. Int. Seminar on Software Visualization, Dagstuhl*, 2002, 151-162.

[4] W. De Pauw, D. Kimelman, J. Vlissides, "Modelling Object-Oriented Program Execution", *Proc. 8th European Conf. on Object-Oriented Programming (ECOOP),* Lecture Notes in Computer Science 821, Berlin: Springer-Verlag, Bologna, 1994, 163-182.

[5] J.P Downey, R. Sethi, R. E. Tarjan, "Variations on the Common Subexpression Problem", *Journal of the ACM. 27(4)*, October 1980, 758-771.

[6] J. Ebert, B. Kullbach, A. Winter, "GraX – An Interchange Format for Reengineering Tools", *Proc. of the 6th Working Conf. on Reverse Engineering (WCRE)*, 1999, 89-98.

[7] P. Flajolet, P. Sipala, J. –M. Steyaert, "Analytic Variations on the Common Subexpression Problem", *Proc. Automata, Languages, and Programming, volume 443 of Lecture Notes in Computer Science, Springer-Verlag*, 1990, 220-234.

[8] A. Hamou-Lhadj, T. C. Lethbridge, "Compression Techniques to Simplify the Analysis of Large Execution Traces", *Proc. International Workshop on Program Comprehension (IWPC),* Paris, 2002, 159-168.

[9] T. C. Lethbridge, "The Dagstuhl Middle Metamodel: A Schema for Reverse Engineering", *(*Electronic Notes in Theoretical Computer Science) (this volume), from a paper presented at *1st International Workshop on Meta-models and Schemas for Reverse Engineering (ATEM),* Victoria, 2003.

[10] R. C. Holt, "An Introduction to TA: The Tuple Attribute Language", *Technical Report, Department of Computer Science, University of Waterloo and University of Toronto*, 1998.

[11] R. C. Holt, A. Winter, A. Schürr, "GXL: Toward a Standard Exchange Format", *Proc. 7^{th} Working Conf. on Reverse Engineering (WCRE)*, 2000, 162-171.

[12] D. Jerding, S. Rugaber, "Using Visualisation for Architecture Localization and Extraction", *Proc. 4th Working Conference on Reverse Engineering*, Amsterdam, Netherlands, October 1997, 56-65.

[13] D. B. Lange, Y. Nakamura, "Object-Oriented Program Tracing and Visualization", *IEEE Computer*, 30(5), 1997, 63-70.

[14] H. A. Müller, K. Klashinsky, "Rigi – A System for Programming-in-the-Large", *Proc. of the Int. Conf. on Software Engineering (ICSE)*, 1988, 80-86.

[15] T. Richner, S. Ducasse, "Using Dynamic Information for the Iterative Recovery of Collaborations and Roles", *Proc. of the 18th Int. Conf. on Software Maintenance (ICSM)*, Montréal, QC, 2002, 34-43.

[16] T. Systä, "Understanding the Behaviour of Java Programs", *Proc. 7th Working Conf. on Reverse Engineering (WCRE)*, Brisbane, QL, 2000, 214-223.