A Metamodel for the Compact but Lossless Exchange of Execution Traces

Abdelwahab Hamou-Lhadj
School of Information Technology and Engineering (SITE),
University of Ottawa
800 King Edward Avenue
Ottawa, Ontario, K1N 6N5 Canada
ahamou@site.uottawa.ca

Timothy C. Lethbridge
School of Information Technology and Engineering (SITE),
University of Ottawa
800 King Edward Avenue
Ottawa, Ontario, K1N 6N5 Canada
tcl@site.uottawa.ca

Abstract

Understanding the dynamics of a software system can be made easier if efficient tool support is provided. Lately, there has been an increase in the number of tools for analyzing execution traces. the need for such tools arises due to the increasing complexity of software. However, these tools have different formats for representing traces, which limits reuse and sharing of data. To allow for better synergies among trace analysis tools, it would be beneficial to develop a standard format for exchanging traces. In this paper, we present a graph-based format, called Compact Trace Format (CTF), which we hope will lead the way towards such a standard. CTF can model traces generated from a variety of programming languages, including both object-oriented and procedural ones. CTF is built with scalability in mind to overcome the vast size of most interesting traces. Indeed, CTF is based on the idea that dynamic call trees can be transformed into ordered acyclic directed graphs by representing similar subtrees only once. An algorithm for on-the-fly generation of CTF-based traces in also provided.

1. Introduction

In [8], we studied the concepts implemented in eight tools for analyzing traces of routine calls. One of the results of this study consists of the fact that these tools use different formats for representing traces, which limits sharing and reuse of data. Although they have common features, each of them has its own advantages and specialized functions. Currently, the only way to take full advantage of these functions is to convert data from one format to another. Writing converters to and from all available formats would be impractical. To address this issue, we have developed an exchange format called CTF (Compact Trace Format) for representing traces of method (routine) calls.

CTF is built with the idea of scalability in mind. It takes advantage of the fact that dynamic call trees can be transformed into ordered directed acyclic graphs (DAG) by representing similar subtrees only once [13, 15, 20]. The original trace can be reconstructed in the normal case where exact matching is used when comparing subtrees. In this default case, CTF is therefore a lossless representation of the trace.

Some of the advantages for having a standard exchange format can be summarised in what follows:

- It reduces the effort required when representing traces.
- It allows researchers to use different tools on the same input, which can help compare the techniques supported by each tool.
- It allows maintainers to combine the techniques from different tools to realize the given maintenance task.

An exchange format consists of two main components [1]: A schema (i.e. metamodel) that represents the entities to exchange and their interconnections, and the syntactic form of the file that will contain the information to exchange.

In this paper, we present the CTF schema and discuss how it is used to represent the information needed to share traces of method calls. We also discuss how CTF data can be ‘carried’ using existing syntactic formats such as GXL (Graph eXchange Language) [11] or TA (Tuple Attribute Language) [10].

This paper is organized as follows: In Section 2, we present an overview of existing metamodels used to describe dynamic information. In Section 3, we discuss the requirements that we used to guide the design of CTF. We present the CTF abstract syntax in the form of a class diagram in Section 4. In Section 5, we discuss the languages that can be used with CTF to represent the data. We present an algorithm for on-the-fly generation of CTF-based traces in Section 6. CTF tool support is the subject of Section 7.

2. Related Work

Several authors have discussed the benefits of representing a dynamic call tree in the form of a directed acyclic graph [13, 15, 20] but none of them attempted to build a metamodel upon this concept.

Riess and Renieris discussed a technique called string compaction that can be used to represent a dynamic call tree as a lengthy string [20]. For example if function 'A' calls function 'B' which in turn calls function 'C', then the sequence could be represented as 'ABC'. Markers are added to indicate call returns. For example, a sequence 'A' calls 'B' and then calls 'C' will be represented as 'ABvC' (the marker 'v' will indicate a call return). However, we posit that this basic representation has a number of limitations: It doesn't support the attachment of various attributes to routines, and can not be easily adapted to support of statement-level traces.

In her master's thesis [16], Leduc presented a metamodel for representing traces of method calls. Her metamodel supports also statement-level traces. Leduc used UML class diagrams to describe the components of the metamodel and the relationships among these components. However, one of the major drawbacks of her approach is the fact that the metamodel is an exact representation of the dynamic call tree. That means that if a trace, for example, contains one million calls then using Leduc's metamodel a trace analysis tool will need to create one million objects. Leduc's metamodel does not take into account any sort of compaction scheme.

There are other trace formats that exist in the literature. However, most of them do not necessarily deal with traces of method calls. In [2], the authors presented STEP, a system designated for the efficient encoding of generalized program trace data. One of the main components of the STEP system is STEP-DL, a definition language for representing STEP traces, which contain various types of events generated from the Java Virtual Machine including object allocation, variable declaration, etc. STEP is useful for applications that explore Java bytecode files. STEP-DL is a specialized metalanguage that describes the structure of the events supported by STEP. The authors argued that a specialized language is a better choice than using a language based on an existing mark-up approach such as XML or SGML. Their first argument is that, as noted by Chilimbi et al. in [3], the wordiness of XML is incompatible with the key compactness requirement for traces. Second, the syntax for document type definitions (DTDs) in such languages tends to be complex for the task at hand.

Hyadas is the Eclipse Test and Performance Project with the aim to “build a generic, extensible, standards-based tool platform” for testing, tracing, profiling, tuning, logging, monitoring, and analysis [12]. Hyades integrates very sophisticated trace collection techniques using dedicated software agents. The Hyades trace model is based on sequential logs of events; it focuses more on trace-to-test conversion and automatic testing instead of program understanding. The CTF model is specifically designed to enable program understanding tools to exchange traces of method calls, which form a natural hierarchy. The DAG based CTF model is not directly supported by the trace model used in Hyades. However, a trace using the CTF model can be built upon information extracted from a trace using the Hyades model. Extra information found in the Hyades model, such as temporal information, correlation between threads, etc. can also supplement the CTF model. The direct benefit is that we will no longer be concerned about trace capture and format conversion and raw trace data persistence, which are provided by the Hyades platform. So these two models are complementary.

Other trace formats such as PDATS [14], HATF [3] and MaStA I/O [21] have been proposed. These formats focus on completely different type of traces. PDATS is a family of trace compression techniques used to compress address and instruction traces, which are commonly used in the performance analysis of computer systems (e.g. simulation of cache memory, pipelined ALUs, load-store units, and other functional units). HATF is a trace format used to represent heap allocation traces (i.e. the events targeted are malloc, free, etc). MaStA I/O focuses on read/write traces that can be used to determine the cost of these statements when applied to databases. All these formats rely on encoding techniques for compressing traces.

3. Requirements for the Design of CTF

Before exploring the design of CTF in more detail, there is a need to have a list of requirements that will help us evaluate its overall quality. Requirements for a standard exchange format have been the subject of many studies [1, 18, 22, 24]. The following goals were used to guide the design of CTF.

3.1 Expressiveness

One important aspect of an exchange format is to support the various types of data that need to be shared. The study we conducted in [8] shows that most trace analysis tools would expect to manipulate traces of method calls at three levels of granularity at least: object, class, and subsystem level.

However, in order to allow non-OO systems to use CTF, we need to permit enough flexibility to represent the necessary constructs that are involved. For example, if the system is written in C then one possible scenario is to include the system files containing the invoked routines.

In addition to this, due to the multi-threaded nature of most existing software systems, the design of CTF needs to consider the threads of execution that are generated from the execution of a given system scenario.

Although this paper is tuned towards program comprehension, we do feel the need to build in the flexibility to represent timestamps, and other statistical information such as the execution time of the routines. We believe that this will enable other types of applications such as profilers to use CTF.

Finally, the maintenance of a software system will typically involve using the static as well as the dynamic aspects of the system. Having said this, we need to make sure that CTF can be used with existing metamodels that capture the static components of the system. In this paper, we show how CTF can be used with the Dagstuhl Middle Metamodel (DMM) [17] to satisfy this requirement although it does not preclude it. DMM is presented in Section 4.6.

3.2 Scalability

The adoption of any exchange format for representing execution traces will greatly depend on the capability to support large-sized information. In [9], we showed that an efficient way for representing traces is to turn them into ordered acyclic directed graphs (DAG). This is because traces contain several repetitions that can be represented only once. Such transformation could reach a compression ratio of 98% [9]. This technique was also used in trace compression and encoding and was shown to scale up to large traces as presented by Reiss [20] and Larus [15].

In addition to this, encoding techniques can also be used to further reduce the physical storage space allocated to traces. For example, one can encode the routine names using special identifiers to avoid dealing with lengthy strings. In [20], Reiss suggests several encoding techniques that aim to compress execution traces such as string compaction, interval compaction, etc.

3.3 Simplicity

The simplicity requirement refers to the simplicity of the CTF specification. What is needed is to have a design that is well documented and not too complex for tool builders to integrate in their tool suites.

3.4 Transparency

This requirement ensures that the information is represented without any alteration. We need to have well-defined mechanisms for generating traces in the form of CTF. The traces will only ever need to be saved as tree structures. To address this requirement, we discuss such mechanism in Section 6.

3.5 Neutrality

This requirement refers to an exchange format that is independent of any specific programming language, platform or technology. This requirement is satisfied due to the fact that the data conveyed in traces of method calls contain almost always the same information independently from the software system from which they were generated. In this paper, we support traces that contain the following information:

- Thread name
- Package or subsystem name
- Class or file name
- Object identifier
- Method name
- Timestamp information
- Method execution time

3.6 Extensibility

The design for extensibility is an important requirement for the design of an exchange format. We address this by an extensive use of abstraction in the design of the CTF schema.

3.7 Completeness

This requirement consists of having an exchange format that includes all the necessary information that is needed during the exchange: the data to exchange as well the schema in which the data needs to be interpreted. The objective is to enable tools to perform checks on the instance data to verify its validity with respect to the schema. We address this requirement by recommending a syntactic form that supports the exchange of the schema as well as the instance data. What is needed is a data 'carrier' that supports the exchange of the metamodel as well as the instance data. GXL and TA both support this requirement.

3.8 Solution Reuse

This requirement consists of reusing some existing technologies in the creation of the new exchange format. This will decrease the amount of time needed for testing the new format. This requirement is one of the main motivations for selecting an existing language for carrying CTF data such as GXL and TA.

3.9 Popularity

The popularity requirement is concerned with the adoption of an exchange format by several users (e.g. tool builders). For this purpose, we have created an API for CTF as well as an eclipse plug-in that will allow different tools to directly generate traces in the CTF format. We have also presented CTF in various conferences. CTF is also supported by a trace analysis tool built at the University of Ottawa and that is called SEAT (Software Exploration and Analysis Tool).

4. CTF Abstract Syntax

Figure 1 shows a UML class diagram that describes the CTF schema. The components of this schema are discussed in the subsequent sections.

4.1 Usage Scenario

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.

4.2 Trace Types

The class Trace is an abstract class that describes common information that different types of traces are most likely to share such as timing information about the generation of the trace, the conditions under which the trace is generated, etc. To create specialized types of traces, one can simply extend this class. Although, we focus on object-oriented systems, we added the RoutineCallTrace class to represent traces of routine calls generated from procedural systems. By routine, we mean any function, whether or not it is a method of a class. Since programming languages such as C++ allow non-method routines (simple C functions) as well as methods, we decided to represent traces of pure method calls as a subclass of RoutineCallTrace. Using this hierarchy, an analyst can create traces of simple functions calls only, traces of method calls only or traces of that combine both, such as C++ execution traces. This design decision is intended to help keep the design simple and understandable.

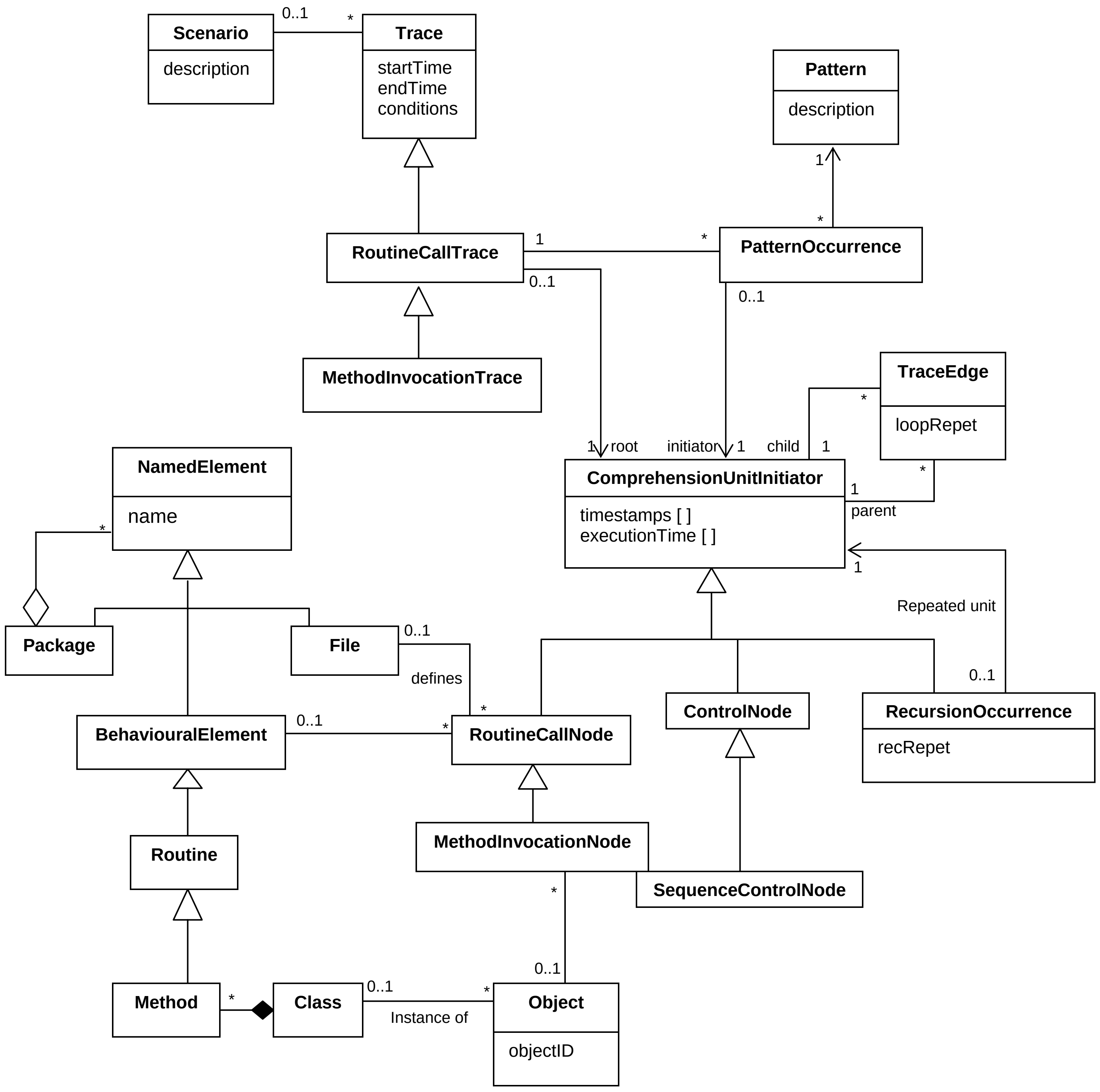


Figure 1. The CTF metamodel

4.3 Nodes and Edges

In our previous work [9], we used the term *comprehension unit* to describe a distinct subtree of the dynamic call tree. The rational behind this is that traces are known to contain many repetitions of the same sequences of events. From the comprehension point of view, a software engineer needs to understand each distinct subtree only once and reuse this knowledge whenever the subtree occurs again in the trace. A detailed discussion about the concept of comprehension units is presented in [9]. In this paper, we use the term *comprehension unit initiator* to refer to the root of a subtree (i.e. the method that initiates a given comprehension unit). Comprehension unit initiators are represented in the class diagram using the ComprehensionUnitInitiator class. These are also 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. Each initiator maintains the timestamps of the method calls it represents.

Edges (i.e. invocations) are represented using the TraceEdge class. An edge is labelled with the number of repetitions due to loops if there are any. We do not use the class name 'Invocation' to allow the model to be extended to include other types of interactions.

A node (ComprehensionUnitInitiator) can either be a routine call node (RoutineNodeCall), a method node (MethodInvocationNode) or a control node (ControlNode). Control nodes represent extra information that might be used by tools to customize portions of the traces. For example, a software engineer exploring traces might need to select a particular subtree and assign to it a description. This can easily be represented by adding a new node to the DAG that holds the description. In this paper, we do not attempt to represent all possible control nodes that might be needed.

4.4 Dealing with Repetitions

One particular control node integrated into CTF is called SequenceControlNode and it is used to represent contiguous repetition of multiple comprehension units. Figure 2 shows how such control node can be used to represent the repetition of the sequence: BC and E.

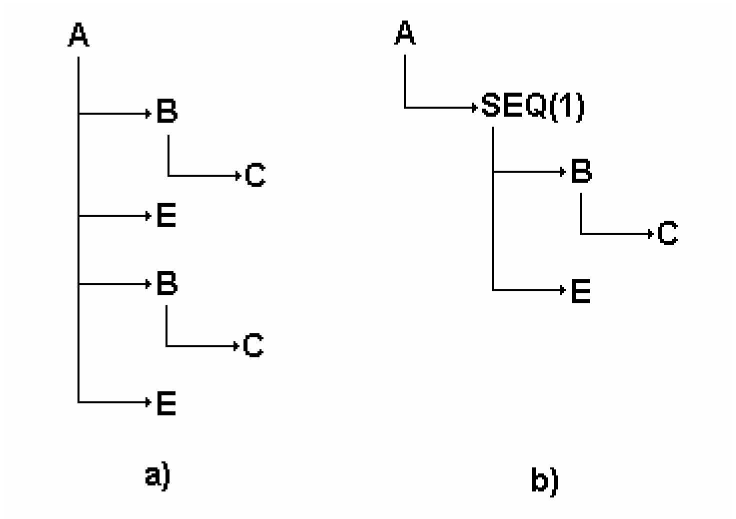


Figure 2. The control node SEQ is used to represent the contiguous repetitions of the comprehension units rooted at B and E

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

4.5 Threads

To represent thread information, we decided to add a class called Thread. Each comprehension unit initiator has an object thread that is associated to it. Threads are identified using unique thread names since this is a common practice in languages such as Java and C++. Leduc made the same design decision [16]. We do not distinguish between thread start/end routines from the other routines for simplicity reasons.

In [20], Riess and Renieris suggest dividing the trace into subtraces where each subtrace corresponds to thread of execution. Timestamp information should permit the reconstruction of the order of execution.

4.6 Static Information

Although the schema presented above 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, Class, and BehaviouralFeature are added and associated to RoutineCallNode to describe references to actual objects representing each class, method, etc. A behavioural feature is a subclass of NamedElement that has an attribute ‘name’. We added the File class to deal with situations where a routine is simply identified using the file where it is defined.

The description of the static components of the system in CTF is similar to the one supported by the Dagstuhl Middle Metamodel (DMM) [17], which is a model for representing the static relationships among the various components of a software system. DMM supports systems developed in most widely used procedural and object-oriented programming languages such as C, C++, and Java. The compatibility between CTF and DMM should enable these two metamodels to work together in the future. Currently, CTF neither requires nor precludes the usage of DMM.

The class ClassObject and the association that links it to MethodInvocationNode allow having traces that describe invocations at the object level. An object identifier is used to uniquely identify objects.

4.7 Behavioural Patterns

Trace patterns (i.e. sequence of events repeated more than once in the trace) are represented using the TracePattern class. This class contains an 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. To capture this information, we use the PatternOccurrence class. This class contains an attribute that describes the number of repetitions of the pattern in a given trace.

4.8 Illustration of CTF

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 3a. The trace involves three classes namely A, B and C and 3 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 3b.

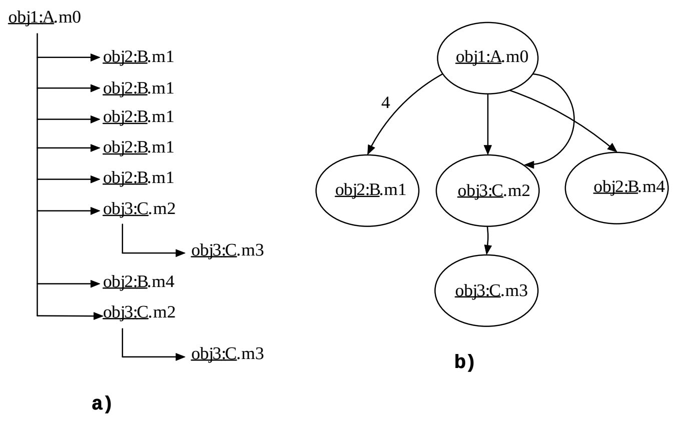


Figure 3. a) An example of a trace as a tree. b) The ordered directed acyclic graph corresponding to this trace

An instance diagram of the above trace using the CTF schema is shown in Figure 4. This diagram omits instances of the static model classes (e.g. Trace, Method, etc) 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 (that are not shown here). Edges are represented using instances of the class TraceEdge. There are 5 edges. The node obj3Cm3 has two incoming edges. The software engineer using the tool can therefore mark this as a pattern, shown here as an instance of PatternOccurrence. The user has indicated using the ‘description’ attribute that this pattern is concerned with the “Drag Mouse” operation. The instance of PatternOccurrence shows which particular trace the pattern occurs in, and which nodes are the initiators (one node in this case).

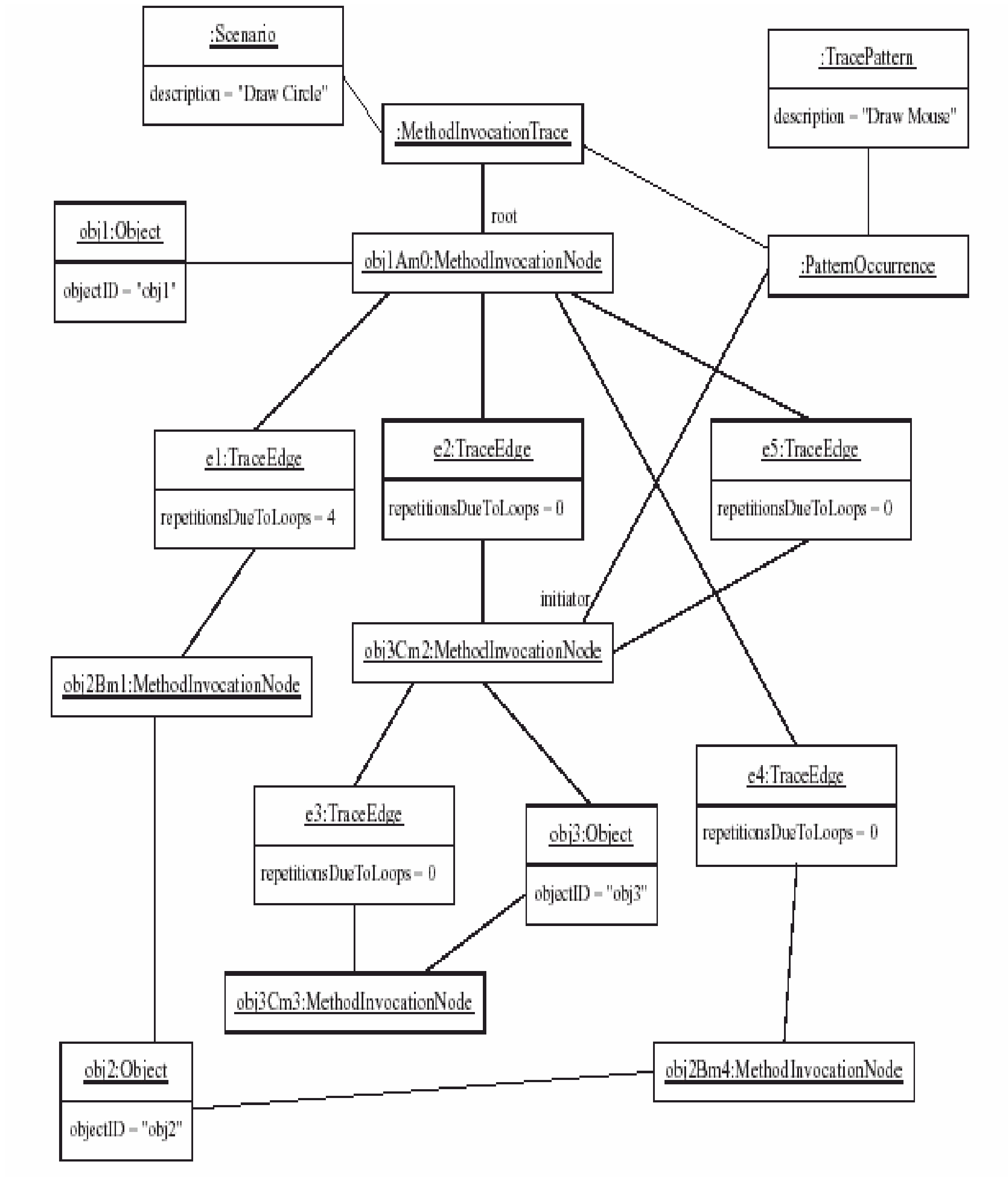

:Scenario
description = "Draw Circle"
:MethodInvocationTrace
:TracePattern
description = "Draw Mouse"
root
obj1:Object
objectID = "obj1"
obj1Am0:MethodInvocationNode
:PatternOccurrence
e1:TraceEdge
repetitionsDueToLoops = 4
e2:TraceEdge
repetitionsDueToLoops = 0
e5:TraceEdge
repetitionsDueToLoops = 0
initiator
obj3Cm2:MethodInvocationNode
obj2Bm1:MethodInvocationNode
e4:TraceEdge
repetitionsDueToLoops = 0
e3:TraceEdge
repetitionsDueToLoops = 0
obj3:Object
objectID = "obj3"
obj3Cm3:MethodInvocationNode
obj2:Object
objectID = "obj2"
obj2Bm4:MethodInvocationNode

Figure 4. An example of a CTF instance data

5. CTF 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 [19]. 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 by representing it using GXL as noted by Brown [2] and Chilimbi et al. [3]. 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.

6. Generating CTF Traces

The objective of this section is to develop an algorithm that aims to generate traces in the form of CTF. Ideally, traces are converted into CTF while they are generated (i.e. during the execution of the system).

The problem of generating CTF based traces can be reduced to the problem of transforming a tree structure into an ordered directed acyclic graph. This problem is also known as the common subexpression problem and was introduced by [5] where a rather complex, linear time solution was given. In [7], Flajolet et al. described a top-down recursive procedure that solves this problem in an expected linear time assuming that the degree of the tree is bounded by a constant [7]. In [23], Valiente presented an iterative version of Flajolet et al's algorithm and applied it to solving the tree matching problem –Finding all occurrences of a subtree t in a tree T.

The algorithm presented by Valiente proceeds by traversing the tree in a bottom-up fashion (from the leaves to the root). Each node is assigned a *certificate* (a positive integer between 1 and n, where n represents the size of the tree). The certificates are assigned in such a way that two nodes *n1* and *n2* have the same certificate if and only if the subtrees rooted at *n1* and *n2* are isomorphic.

To compute the certificates, the algorithm uses a signature scheme that identifies each node. The signature of a node *n* consists of the concatenation of its label and the certificates of its direct children, if there are any. For example according to Figure 5, the signature of A should be "A 1 2". The signature of a leaf is simply its label (e.g. the signature of C is "C"). A global hash table is used to store the certificates and signatures and ensure that similar subtrees will always hash to the same element.

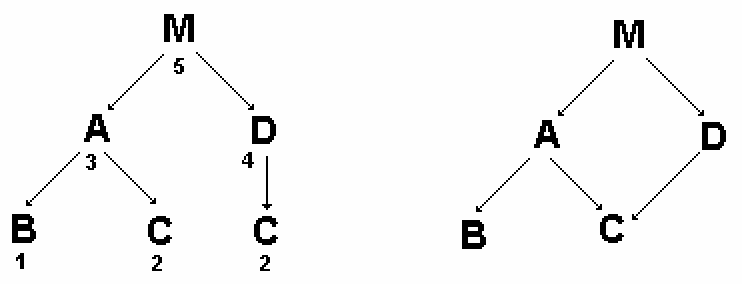


Figure 5. A tree (left) and its transformed DAG (right). The integers correspond to the certificates of the nodes

The global table that results from applying Valient's algorithm to the tree of Figure 5 is shown in Table 1. From this table, we can easily construct the ordered directed acyclic graph that corresponds to the tree of Figure 5. The root of the graph corresponds to the table entry with the highest certificate.

Table 1. Result of applying Valiente's algorithm to the tree of Figure 5.

Signature	**Certificate**
B	1
C	2
A 1 2	3
D 2	4
M 3 4	5

The complexity of the algorithm consists of the time it takes to traverse the tree, the time it takes to compare two subtrees, and the time it takes to compute the signatures.

Valiente's algorithm can easily be adapted to generate CTF traces which are already saved in the form of tree structures. In this thesis, we improve the algorithm to consider transformation of the trace as the events are generated (i.e. on the fly). The second improvement takes into account the various matching criteria that can be used to consider two subtrees as similar but not necessarily isomorphic.

6.1 The Adaptation of Valiente's Algorithm

The idea behind the algorithm is to be able to detect when subtrees are constructed during the generation of the trace. Once this is done, we can compute their signatures, check if they exist in the global table, and update the global table. The final table will represent the DAG version of the trace.

To develop the algorithm, let us consider the following:

- The trace is generated as a sequence of events that have the following form: (Label, Nesting level). The label can be composed of the thread name, subsystem name, class name, object identifier, and the method name. The nesting level indicates the nesting level of the routine calls. The root of the dynamic call tree has a nesting level equal to zero. Other information can be added, such as timestamps, execution time of a routine, the number of statements, etc. For simplicity reasons, we do not include these in the algorithm.

- We will need to create a tree structure that will temporarily hold nodes corresponding to the trace events. These nodes will be destroyed once their signatures are computed.

- A node of the tree will contain the following attributes:

label	This is the label of the node.
parent	This represents the parent of the node.
nl	This represents the nesting level of the node (this will avoid computing it every time we need to refer to the nesting level)

certificate	This represents the certificate that will be assigned to the node after checking the global table. Initially *certificate = 0*.

The steps of the algorithm are:

```
1.  Read the first event e = (label, nl)
2.  Create a node called root that represents the root of the temporary tree: root.label = e.label;
    root.parent = null; root.nl = 0; root.certificate = 0
3.  prevNode = root   // Keeps track of the previous node
4.  certificate = 1
5.  globalTable = A Hash Table: The keys represent signatures and the values represent certificates.
                  We will use the put and get functions to insert and retrieve elements from
                  globalTable
6.  For every event $e_1$ = ( $label_1$, $nl_1$ ) Do
      6.1 While ( $e_1.nl_1$ <= previousNode.nl ) Do

          /*  If the current nesting level ($e_1.nl_1$) if less than or equal to the nesting level of
              the previous node which is here represented by parent.nl then the node
              prevNode must be a subtree. In fact, all the parents of the node prevNode that
              have a nesting level that is greater than or equal to $nl_1$ must also form
              subtrees. This explains why we need to use a loop to check for formed
              subtrees. */

          6.1.1  signature = Signature of  prevNode
          6.1.2  If  globalTable contains the key signature Then
                       a.  prevNode.certificate = globalTable.get (signature)
                 Else
                       b.  globalTable.put (signature, certificate)
                       c.  prevNode.certificate = certificate
                       d.  certificate++   // update the certificate
                 End If
          6.1.3  prevNode = prevNode.parent
        End While
      6.2 If there are nodes for which certificates have been assigned then delete these nodes
          except the one that has the same nesting level as $nl_1$.
      6.3 Create a new node called node that represents $e_1$: node.label = $e_1$.label; node.parent =
          prevNode; node.nl = $e_1.nl_1$; node.certificate = 0
    6.4 prevNode = node
  End For
7. Compute the signatures and check the table for all nodes from the last event to the root. Once
   this is done, delete all remaining nodes.
```

We illustrate the steps of the algorithm using the tree below:

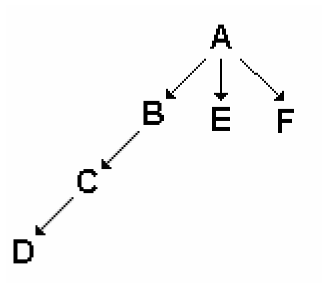
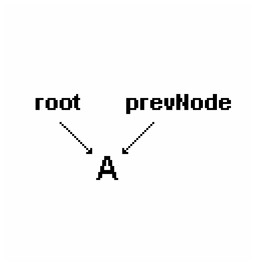
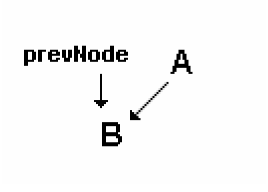
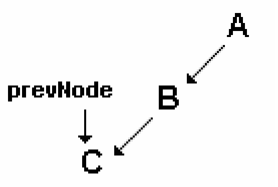
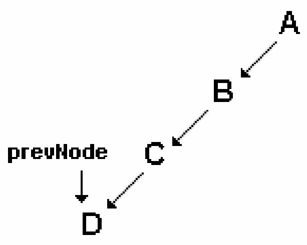
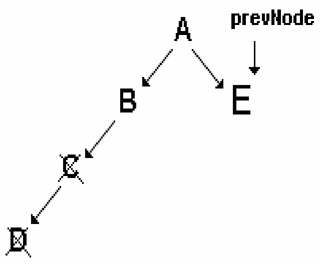
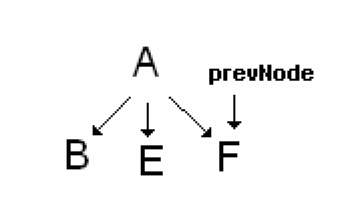
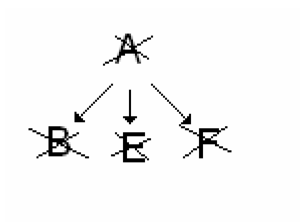


Figure 6. A sample tree that will be used to illustrate the algorithm

Steps of the algorithm	**Illustration**	**Global Table**
Step 1 consist of reading the first event of the trace (i.e. *e = (A, 0)*) Step 2 creates a root node that corresponds to *e = (A, 0)* Step 3 to 5 initialise variables to keep track of: - The previous node (*prevNode*) - The certificate value (*certificate*) - The global table (*globalTable*)		Signature | Certificate
In Step 6, the algorithm reads the next events one by one Step 6.1 checks if by encountering B, we have discovered that A was the root of a complete subtree. We do this by comparing B's nesting level to that of A. Since B has a higher nesting level, the algorithm goes to Step 6.2: checks if there are nodes for which certificates have been assigned in		Signature | Certificate

order to clean up memory. This is not the case, 6.3 creates a node for B and links A to B. 6.4 updates prevNode.		
The algorithm continues in Step 6 to deal with the event (C, 2). It again checks whether a subtree has been formed at Step 6.1. Since this is not the case, the algorithm goes to 6.2, checks if there are nodes for which certificates have been assigned in order to clean up memory. This is not the case, 6.3 creates a node for C and links B to C. 6.4 updates prevNode.		Signature | Certificate
Same thing with the event (D, 3).		Signature | Certificate
The algorithm encounters the event (E, 1). In 6.1, the comparison reveals that the nesting level of E is less than the nesting level of D, which means than D is the root of a fully formed subtree. Therefore, the algorithm computes the signature of D (which is equal to "D" in this case, since D is a leaf), checks if the signature exists in the table, if not, assigns to D the current certificate (i.e. certificate = 1), inserts the pair (signature of D, 1) into the global		Signature | Certificate D | 1 C 1 | 2 B 2 | 3

table, assigns the certificate to the node that holds D, and updates the value of *certificate* (steps b, c, and d) Step 6.1 causes the algorithm to repeat the process with the node C. C's signature is 'C 1' and its certificate is 2. It repeats the process again with B. B's signature is 'B 2' and its certificate is 3. Step 6.2 deletes the nodes corresponding to the nodes D and C. However, it does not delete the node B since we will need it to compute the certificate of A In 6.3, the algorithm creates a node for E, links A to E and updates prevNode		
The event (F, 1) occurs causing the algorithm to compute the signature of the node E, to find that it doesn't exist in the table, to assign it a certificate, and to update the table (Steps b, c, d)		Signature — Certificate D — 1 C 1 — 2 B 2 — 3 E — 4
No more events are read. The algorithm (Step 7) therefore computes the signatures of the last node all the way to root (i.e. nodes F and A), checks whether the signatures already exist, if not assigns new certificates to these nodes, and update the table. All remaining nodes are then deleted		Signature — Certificate D — 1 C 1 — 2 B 2 — 3 E — 4 F — 5 A 3 4 5 — 6

The resulting table represent the ordered directed acyclic graph. The root of the graph is the entry that has the highest-numbered certificate, which is in this case the node labelled 'A'. The links from one node to another are indicated in the signatures.

Generating CTF-based traces will need to take into account the syntactic language used to describe the data such as GXL, TA, etc.

6.2 Matching Criteria

Although CTF Many authors have suggested matching criteria that can be used to group similar sequences of events as instances of the same pattern [4].

Matching subtrees using class names, package names, etc:

One way of generalizing the algorithm is to group similar sequences of calls according to whether they invoked objects of the same class, same package, etc. To adapt the algorithm to such situations, we need to be able to select part of the node labels that will be used during the comparison process. The most restrictive case will be the ones where all constituents of the label are compared.

Ignoring the number of repetitions:

Ignoring the number of repetitions of contiguous sequences of calls when looking for similar subtrees can be applied to avoid having two subtrees that differ only because some elements are repeated more than others as shown in Figure 7.

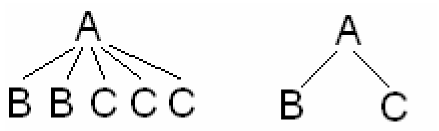


Figure 7. If contiguous repetitions are ignored then the above subtrees will be considered similar

Let us consider that the certificates of B and C are 1 and 2 respectively. In this case, the computation of the signature of A should ignore the number of times B and C are repeated in the subtree on the left side of Figure 7. This means that the signature of A should be equal to “A 1 2” in both cases.

Ignoring the order of calls:

If applied to the subtree of Figure 8, the ‘ignoring order of calls’ matching criterion will consider these subtrees as equivalent.

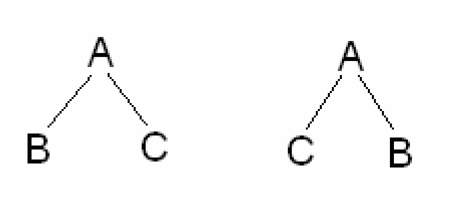


Figure 8. If order of calls is ignored then the above subtrees will be considered similar

To generalize the algorithm to unordered trees, we need to sort the certificates that appear in the signatures and then proceed with comparing the signatures. For example, if B’s certificate is 1 and that C’s certificate is 2 then A’s signature should be in both cases “A 1 2” (i.e. the certificates are sorted).

7. Tool Support

CTF is the exchange format used by a trace analysis tool built in the University of Ottawa called SEAT (Software Exploration and Analysis Tool). The tool manipulates traces in CTF (Compact Trace Format) and displays them using a tree widget. To help the user extract useful information from a trace, SEAT implements several trace compression techniques such as the detection of utilities, application of matching criteria, detection of patterns, etc. An analyst can compress a trace to the level where he or she can understand important aspects of their structure. A CTF API has also been created. It contains the main functions to create and manipulate CTF components.

8. Conclusions and Future Work

A common exchange format is important for allowing different tools to share data. In this paper, we presented CTF (Compact Trace Format) a schema for representing traces of routine calls.

To deal with the vast size of typical traces, we designed CTF based on the idea that dynamic call trees can be turned into ordered directed acyclic graphs, where repeated subtrees are represented only once.

CTF supports traces defined at different levels of abstraction including object, class and package level. It also supports the specification of threads of execution. Additional information such as timestamps and routine execution time are added to enable profilers to use CTF.

CTF, as described in this paper, is a schema. Trace data conforming to CTF can be expressed using GXL, TA, or any other existing data 'carrier' language. However, we suggest using a compact representation since doing otherwise would somewhat defeat the compactness objective of CTF.

An algorithm for the on the fly generation of CTF-based traces was also presented. We showed how this algorithm can easily support various matching criteria; such criteria can be used to consider similar but not necessarily identical subtrees.

CTF is lossless, i.e. the original trace can be reconstructed, when the simplest matching criterion is used: Two sequences of any length of calls to the same routine are considered identical.

Future work should focus on three aspects: The first aspect consists of experimenting with numerous execution traces of many systems to further evaluate 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. Although CTF is now supported by a trace analysis tool called SEAT, we still 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-process communication.

References

[1] T. Bowman, M. W. Godfrey, and R. C. Holt, “Connecting Architecture Reconstruction Frameworks”, *In Proc. of the 1st International Symposium on Constructing Software Engineering Tools (CoSET’99)*, Los Angeles, CA, 1999, pp 43–54

[2] R. Brown, K. Driesen, D. Eng, L. Hendren, J. Jorgensen, C. Verbrugge, and Q. Wang, **“**STEP: a framework for the efficient encoding of general trace data”**,** *Workshop on Program Analysis for Software Tools and Engineering*, South Carolina, USA, 2002, pp. 27 - 34

[3] T. Chilimbi, R. Jones, and B. Zorn, “Designing a trace format for heap allocation events”, *In Proc. of the ACM SIGPLAN International Symposium on Memory Management (ISMM)*, Minneapolis, MN, USA, Oct. 2000, pp. 35-49

[4] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, J. Yang, “Visualizing the Execution of Java Programs”, *In Proc. International Seminar on Software Visualization, Dagstuhl,* 2002, pp. 151-162

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

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

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

[8] Hamou-Lhadj and T. Lethbridge, “A Survey of Trace Exploration Tools and Techniques”, *In Proc. of the 14th Annual IBM Centers for Advanced Studies Conferences (CASCON)*, IBM Press, Toronto, Canada, October 2004, pp.42-55

[9] Hamou-Lhadj and T. Lethbridge, “Measuring Various Properties of Execution Traces to Help Build Better Trace Analysis Tools”, *To appear in the 10th IEEE International*

Conference on Engineering of Complex Computer Systems, Shanghai, China, June 2005

[10] R. C. Holt, "An Introduction to TA: The Tuple Attribute Language", *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. 7th Working Conference on Reverse Engineering (WCRE)*, 2000, pp. 1962-171

[12] Hyades Project: http://www.eclipse.org/tptp/

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

[14] E. E. Johnson, J. Ha, and M. Baqar Zaidi, "Lossless Trace Compression", *IEEE Transactions on Computers*, 50(2):158{173, Feb. 2001

[15] J. R. Larus, "Whole program paths", *In Proc. of the ACM SIGPLAN '99 Conference on Programming language design and implementation*, Atlanta, United States, ACM Press, 1999, pp. 259-269

[16] J. Leduc, "Towards Reverse Engineering of UML Sequence Diagrams of Real-Time, Distributed Systems through Dynamic Analysis", Master Master of Applied Science, Carleton University, 2004

[17] T. C. Lethbridge, "The Dagstuhl Middle Model: An Overview", *In Proc. of the First International Workshop on Meta-models and Schemas for Reverse Engineering (ATEM)*, Victoria, Canada, 2003

[18] T. C. Lethbridge and N. Anquetil, "Architecture of a Source Code Exploration Tool: A Software Engineering Case Study", *Computer Science Technical Report TR-97-07*, University of Ottawa, Ottawa, Canada, November 1997

[19] H. A. Müller, K. Klashinsky, "Rigi – A System for Programming-in-the-Large", *In Proc. of the International Conference on Software Engineering (ICSE)*, 1988, pp. 80–86.

[20] S. P. Reiss, and M. Renieris, "Encoding program executions", *In Proc. of the 23rd international conference on Software engineering*, Toronto, Canada, 2001, pp. 221-230

[21] S. J. G. Scheuerl, R. C. H. Connor, R. Morrison, J. E. B. Moss, and D. S. Munro, "The MaStA I/O trace format", *Technical Report CS/95/4*, School of Mathematical and Computational Sciences, University of St Andrews, North Haugh, St Andrews, Fife, Scotland, 1995

[22] G. St-Denis, R. Schauer, and R. K. Keller, "Selecting a Model Interchange Format: The SPOOL Case Study". *In Proc. of the 33rd Annual Hawaii International Conference on System Sciences*, Maui, Hawaii, January 2000

[23] G. Valiente, "Simple and Efficient Tree Pattern Matching", *Research report*, *Technical University of Catalonia*, E-08034, Barcelona, 2000

[24] S. Woods, S. J. Carrière, and R. Kazman, "A semantic foundation for architectural reengineering and interchange", *In Proc. of International Conference on Software Maintenance (ICSM '99)*, Oxford, England, August 1999, pp. 391–398