Towards a Scalable Representation of Run-Time Information: The Challenge and Proposed Solution

Abdelwahab Hamou-Lhadj
Electrical and Computer Engineering, Concordia University, 1455 de Maisonneuve West, Montreal, Canada
abdelw@ece.concordia.ca

Keywords: Application modernization, software maintenance, reverse engineering, metamodeling, execution traces.

Abstract: An important issue in application modernization is the time and effort needed to understand existing applications. Understanding the dynamic aspects of software is a difficult task, further complicated by the lack of a scalable way of representing the extracted knowledge. The behavior of a software system is typically represented in the form of execution traces. Traces, however, can be extraordinary large. Existing metamodels such as the Knowledge Discovery Metamodel and the UML metamodel provide limited support for handling large execution traces. In this paper, we describe a metamodel called the Compact Trace Format (CTF) for efficient modeling of traces of routine (method) calls generated from multi-threaded systems. CTF is intended to facilitate the interoperability among modernization tools that focus on the analysis of the behavior of software systems. CTF is designed to be easily extensible to support other types of traces.

1 INTRODUCTION

Software modernization is defined as the process of understanding and evolving existing software systems for the purpose of improving their various facets (Bézivin 2004).

Understanding a software system requires both static and dynamic analysis techniques. The former focuses on exploring the structure of the system by analyzing its source code, whereas the latter, which is the focus of this paper, provides insight into the system's behavioral properties. Both approaches aim at extracting knowledge about the system under study that can later be fed to a modernization tool for further analysis.

There exist several standards for representing the knowledge extracted from software systems, among which the most recent is the Knowledge Discovery Metamodel (KDM) (KDM 2007) supported by the Object Management Group (OMG). Others include the UML 2 metamodel (UML 2005), the Dagstuhl Middle Metamodel (DMM) (Lethbridge 2003), etc.

However, while existing metamodels provide a full range of constructs for the modeling of the static aspects of the system (i.e. its static components and the way they interact), they do not support efficient representation of the system's behavioral features, typically represented in the form of execution traces. Traces have historically been difficult to work with since they may contain millions of events. The challenge is to develop a trace format that scales up to handle large and complex traces.

In this paper, we describe ongoing work towards the development of a scalable format for representing execution traces. A particular format called CTF (Compact Trace Format) is proposed. CTF focuses on representing traces of routine (methods) calls generated from multi-threaded systems. It is built with scalability in mind using graph theory concepts. In addition, CTF metamodel is defined in such a way that it can be easily extended to support other types of traces and constructs.

The remaining of this paper is organized as follows: In Section 2, we discuss the requirements that guided us in the development of CTF. In Section 3, we present the CTF metamodel and its syntactic form. CTF semantics are presented in the appendix section. In Section 4, we discuss related studies and how they differ from the work presented in this paper.

2 REQUIREMENTS FOR A TRACE EXCHANGE FORMAT

Requirements for an effective exchange format have been the subject of many studies (Bowman et al. 1999, St-Denis et al. 2000, Woods et al. 2000). In this section, we present the ones that a trace exchange format should fulfill in order to facilitate its adoption. These requirements were also used to guide the development of CTF.

2.1 Scalability

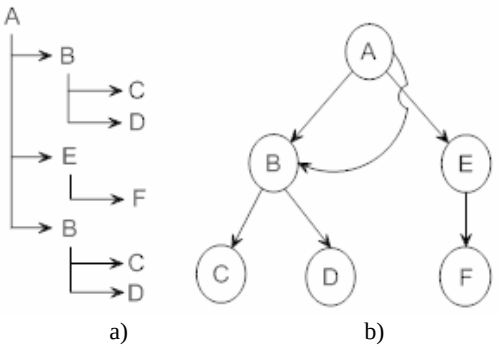
The adoption of a trace exchange format will greatly depend on its ability to support large-sized traces. Trace, however, tend to contain many repetitions due to the presence of loops and recursions in the source code. They also contain non-contiguous repetitions known as trace patterns. The scalability of a trace exchange format can, therefore, be significantly improved if repeated events are factored out and represented only once. For example, the trace of routine calls described in Figure 1a can be transformed into an ordered directed acyclic graph (DAG) represented in Figure 1b by representing the repeated subtree rooted at "B" only once. This technique was first introduced by Downey et al. in (Downey et al. 1980) to improve tools that manipulate trees. It was also used by Larus (Laurs 1999) and Reiss and Renieris (Reiss and Renieris 2001) to encode traces with the objective of saving disk space.

It should be noticed, however, that the graph must be *ordered* in order to be able to restore the initial order of calls. In addition, the graph representation of a trace does not necessarily result in a loss of information associated with individual nodes of the tree such as timing information commonly collected when generating a trace. The simplest solution is to augment the nodes of the graph with ordered collections that holds the information describing individual nodes of the tree.

In previous work (Hamou-Lhadj and Lethbridge 2005), we applied this compaction technique to over thirty execution traces and showed that it can reach a 97% compression ratio (i.e. the graph contains only 3% of the number of nodes of the tree). This has led us, as we will see in Section 3, to build CTF using the ordered DAG as its main mechanism.

2.2 Completeness

This requirement consists of having an exchange format that includes all the necessary information needed during the exchange: the data to exchange as well the metamodel that describes the structure of the data. The rationale behind this is to enable tools to check the validity of the instance data against the metamodel. To address this requirement, we need to select a syntactic form language (i.e. the language that "carries" CTF instance data) that is designed to support the exchange of the instance data as well as the metamodel. In Section 3.2, we discuss possible syntactic forms that could be used with CTF.



a) b)

Figure 1: a) An example of a call tree. b) The compact form of the call tree

2.3 Extensibility

The data model of a trace exchange format must be flexible enough to support new types of traces, language specific entities and properties, and some properties that are specific to the trace analysis tool that uses the format. CTF addresses this requirement by adopting an open design based on abstract classes that allow new constructs to be easily added.

2.4 Tool Support

In order to facilitate the adoption of an exchange format by tool builders, we need to develop well-defined mechanisms that facilitate the manipulation of traces. First, we need to design procedures that ensure that the information exchanged is represented without any alteration. Second, we need to develop algorithms for on the fly generation of traces in the new format. Finally, we need to create converters that will convert other commonly used formats into the new format to facilitate the transition to the new exchange format. Since CTF is still an ongoing project, this requirement will be addressed in future work.

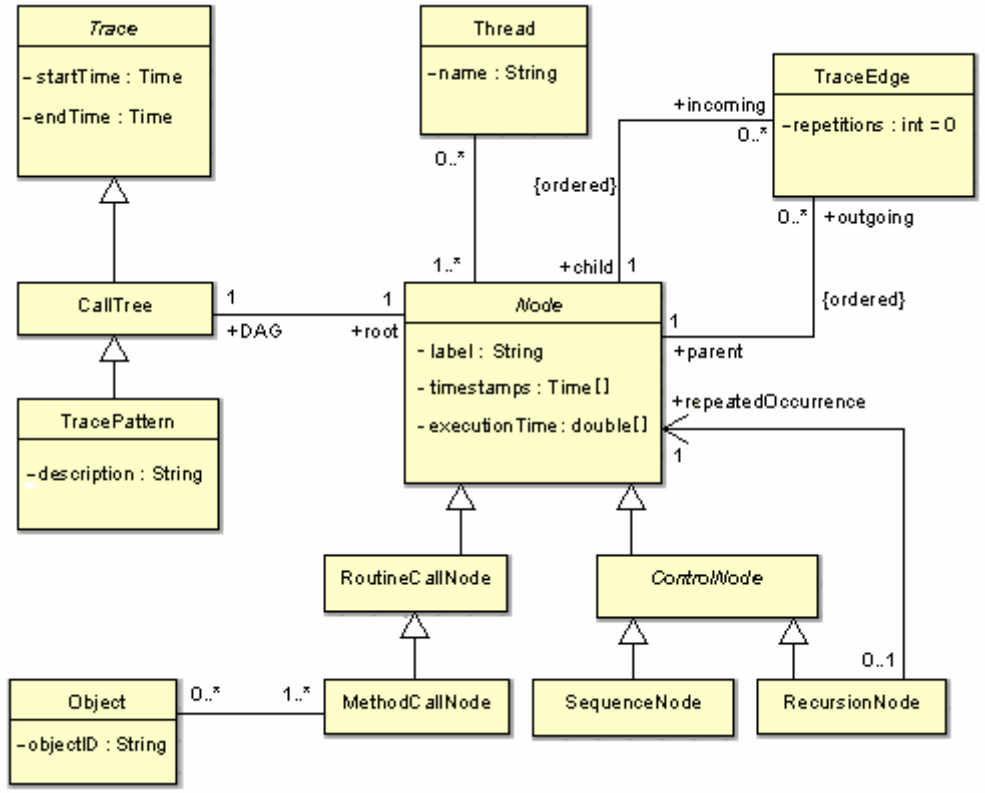


Figure 2: CTF Metamodel

3 CTF

In this section, we present CTF metamodel and its syntactic form. CTF semantics are presented in the appendix section.

The initial version of the CTF metamodel was presented at the First International Workshop on Metamodels and Schemas for Reverse Engineering (Hamou-Lhadj et al. 2003). The metamodel had since undergone significant changes. The current version of CTF is described in this paper. Due to limited space, we did not attempt to show the changes made to the previous version.

3.1 CTF Metamodel

CTF metamodel is presented in Figure 2. CTF models traces as ordered directed acyclic graphs and not as tree structures. The elements of CTF metamodel as presented in what follows:

The class Trace is an abstract class that describes common information that different types of traces usually share such as timing information. To create other types of traces, one needs to extend this class.

The class CallTree depicts a trace of routine or method calls. The class Node refers to nodes in the ordered DAG; each can have many child nodes and many parent nodes as illustrated on the diagram using the parent and child roles. Each node maintains a collection of timestamps and another one that stores the execution times of the individual routine calls represented by this node.

Edges (i.e. calls) are depicted using the TraceEdge class. An edge is labeled with the number of repetitions (set to zero by default) that may exist due to loops or recursion as illustrated in Figure 3. In this example, the number of repetition of B is two (i.e. the number of times B appears in this trace is three).

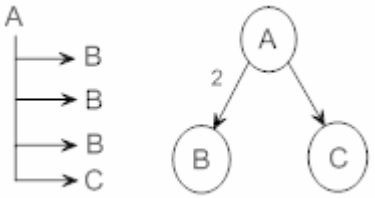


Figure 3: Contiguous repetitions collapsed and replaced by the number of repetitions.

As its name indicates, the class TracePattern represents patterns of execution invoked in the trace. A trace pattern is defined as a sequence of events that is repeated non-contiguously in the trace. They are used by software engineers as a means to uncover key domain concepts from the trace. The common hypothesis is that the same sequence of calls that appear in various places in the trace might encapsulate some knowledge about the system such as a particular aspect of an algorithm, etc. (Jerding and Rugaber 1997, Systä 2000). Software engineers often associate a textual description to trace patterns that are deemed important. The class TracePattern uses an attribute called "description" to capture this textual description.

A node can either be a routine call node (RoutineCallNode), a method node (MethodCallNode), or a control node (ControlNode). A RoutineCallNode object represents a call to a routine that is not a method of a class, whereas a MethodCallNode stands for a method invocation. Method calls are considered as routine calls except that they may contain additional information such as the objects (represented using the class Object) on which the methods are invoked.

The above metamodel relies on a string label to identify trace events (i.e. the routines invoked). It does not provide any other static information about the routines (or methods) invoked such as the parameter list and return type. The reason is that CTF is intended to work with existing metamodels such the UML metamodel, KDM, or DMM. For example, DMM defines two classes called Routine and Method that describe the static elements of a routine and a method respectively. Assuming that CTF is used with DMM, then the classes RoutineCallNode and MethodCallNode will need to be linked to DMM classes Routine and Method in order to retrieve static information. This linkage is not defined in this paper and will be the subject of future improvements of CTF. The advantage of such design is that it allows CFT to be adopted by static analysis tools that use the aforementioned metamodels.

Control nodes represent extra information that might be used by software engineers to customize parts of the trace. In particular, we define two control nodes: SequenceNode and RecursionNode. A SequenceNode object is used to represent contiguous repetitions of multiple sequences of events. Figure 4 illustrates how such node is used to avoid repeating the sequence "BC and D" twice. The label "SEQ" is used in CTF to identify a SequenceNode object.

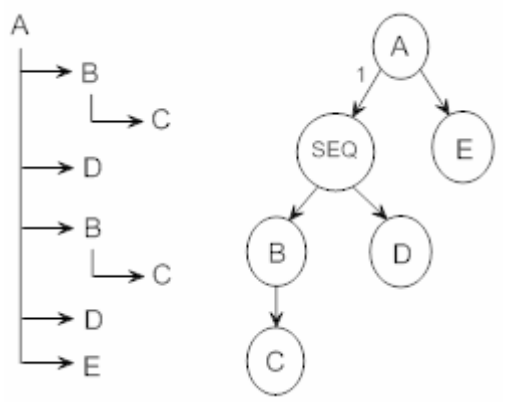


Figure 4: The control node SEQ is used to represent the contiguous repetitions of the subtrees rooted at B and D.

A recursive sequence of calls is represented using another control node called RecursionNode, which in turn refers to the recursively repeated sequence of calls. Figure 5 shows a recursion occurrence node labeled "REC" that is used in CTF to collapse the recursive repetitions of the node B. Note that the number of repetitions is captured through the attribute "repetitions" of the class TraceEdge since an edge will be created between the routine "A" and the recursive sequence.

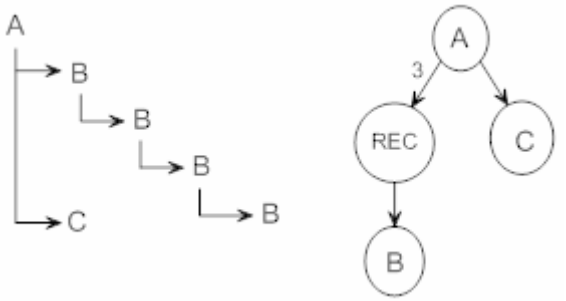


Figure 5:.The control node REC is used in CTF to represent the recursive repetitions of a sequence of calls.

The class Thread represents the thread that executes the corresponding portion of the trace (represented by the class Node). Threads are identified using unique thread names since this is a common practice in languages such as Java and C++. We do not distinguish between thread start/end routines from other routines for simplicity reasons. An obvious extension to this model is to fully support various multi-thread communication mechanisms.

3.2 Syntactic Form

CTF instance data can be carried using a syntactic form that supports the representation of graph structures. There exist several languages that satisfy this requirement, which differ essentially on whether they rely on XML or not. In this paper, we discuss how the Graph Exchange Language (GXL) (Holt et al. 2000) and the Tuple Attribute (TA) (Holt 1998) can be used with CTF. The choice of these languages is due to the fact that they are widely used by the reverse engineering research community. In addition, both languages support the exchange of the instance data as well as the metamodel; this is compliant with the completeness requirement discussed in Section 2.2.

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 (Ebert 1999), TA (Holt 1998), and RSF (Müller 1988). GXL has been widely adopted as a standard exchange format for various types of graphs by both industry and academia.

However, an XML-based representation of a trace would tend to be much larger than necessary due to the use of XML tags and the explicit need to express the data as XML nodes and edges. The compactness benefits of a trace exchange format discussed earlier would therefore be partially cancelled out by representing it using XML. As noted by Chilimbi et al. (Chilimbi et al. 2000), whereas the wordiness of XML would not be a problem when expressing moderately sized structures in other domains, the sheer hugeness of traces suggests an alternative might be appropriate. In addition, a general XML format often requires more processing than a tuned special format. XML will ultimately be parsed and processed to create an internal model. Performance of parsing XML poses an obstacle, especially for large data set. One study related to the performance of XML parsing showed that XML loading is 26 times slower than flat file with delimited format (Nicola and John 2003). The situation is worsened the metamodel validation step is taken into account.

One reasonable alternative to GXL is the Tuple Attribute (TA) (Holt 1998), which would substantially reduce the space required by a CTF trace (since it is not based on XML). The TA language was originally developed to help visualize information about software systems. It has been used as a model interchange format in several contexts and has a reasonable tool support despite the fact that it is not XML-based.

We are still in the process of testing CTF in order to determine the most suitable syntactic form that represents its instance data efficiently.

4 RELATED WORK

As previously mentioned, the Knowledge Discovery Metamodel (KDM) was introduced by OMG to allow interoperability among software modernization tools (KDM 2007). KDM supports a large spectrum of software artifacts at various levels of abstraction. The KDM package dedicated to the modeling of the behavioral aspects of a software system is called Action Package. The package describes how various types of a system's execution are modeled such as control flows, traces of routine calls, etc. However, KDM's approach to representing traces of routine calls is an exact representation of the dynamic call tree. That means that if a trace, for example, contains one million calls then, using the KDM metamodel, a modernization tool will need to create one million objects in memory. KDM does not take into account any sort of compaction scheme.

The UML 2 metamodel is another alternative (UML 2005). However, it is designed for forward engineering, and omits the compact representation of traces as well. It is also rather more complex than what appears to be needed for simple modernization tools that are tuned toward the analysis of execution traces.

In her master's thesis (Leduc 2004), Leduc presented a metamodel for representing traces of method calls, which supports also statement-level traces. Similar to KDM and the UML metamodel, Leduc's approach does not include any compaction scheme.

Reiss and Renieris proposed a technique called string compaction that can be used to represent a dynamic call tree as a lengthy string (Reiss and Renieris 2001). 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 does not have the flexibility to attach various attributes to routines, and cannot be easily adapted to support other kind of traces such as statement-level traces.

In (Brown et al. 2000), the authors presented STEP, a system designated for the efficient encoding of 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 and does not explicitly deal with traces of routine calls.

CONCLUSIONS

Application modernization requires the representation of knowledge about the system under study. While existing metamodels such as KDM, UML, and DMM capture efficiently the static aspects of software, they lack an efficient representation of the behavioral aspects of the system. In this paper, we presented CTF (Compact Trace Format), an exchange format for representing traces of routine (method) 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 factored out. CTF, as described in this paper, is a metamodel. Trace data conforming to CTF can be expressed using GXL, TA, or any other data "carrier" language. However, we suggest using a compact representation since doing otherwise would somewhat defeat the compactness objective of CTF.

While CTF covers a significant gap in terms of exchanging traces of routine calls, dynamic analysis is a highly versatile process that has a large number of needs including needs for dynamic information that is not necessarily supported by CTF. We need to work towards enhancing CTF in order to support other types of traces and constructs.

The main future direction is to agree on what should be represented in a trace exchange format. We also need to investigate how various traces can be compacted so as the embedded repetitive behavior is represented only once. Moreover, we need to analyze the proper syntactic form that is most suitable for representing lengthy traces.

Finally, we need to work towards the adoption of CTF by developing converters that can easily convert existing formats into CTF. We also need to develop mechanisms for the on the fly generation of CTF traces.

REFERENCES

Bézivin J., 2004. Model Engineering for Software Modernization. In *WCRE'04, 11th Working Conference on Reverse Engineering.*

Bowman T., Godfrey M. W., and Holt R. C., 1999. Connecting Architecture Reconstruction Frameworks. In *CoSET'99, 1st International Symposium on Constructing Software Engineering Tools*.

Brown R., Driesen K., Eng D., Hendren L., Jorgensen J., Verbrugge C., and Wang Q., 2002. STEP: a framework for the efficient encoding of general trace data. In *PASTE'02, Workshop on Program Analysis for Software Tools and Engineering*.

Chilimbi T., Jones R., and Zorn B., 2000. Designing a trace format for heap allocation events. In *ISMM'00, ACM SIGPLAN International Symposium on Memory Management.*

Downey J. P., Sethi R., Tarjan R. E., 1980. Variations on the Common Subexpression Problem. *Journal of the ACM. 27(4)*.

Ebert J., Kullbach B., Winter A., 1999. GraX – An Interchange Format for Reengineering Tools. In *WCRE'99, 6th Working Conference on Reverse Engineering*.

Hamou-Lhadj A. and Lethbridge T., 2003. The Compact Trace Format, In *ATEM'03, 1st International Workshop on Metamodels and Schemas for Reverse Engineering.*

Hamou-Lhadj A. and Lethbridge T., 2005. Measuring Various Properties of Execution Traces to Help Build Better Trace Analysis Tools. In *ICECCS'05, 10th IEEE International Conference on Engineering of Complex Computer Systems.*

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

Holt R. C., Winter A., Schürr A., 2000. GXL: Toward a Standard Exchange Format. In *WCRE'00, 7th Working Conference on Reverse Engineering*.

Jerding D., Rugaber S., 1997. Using Visualisation for Architecture Localization and Extraction. In *WCRE'97, 4th Working Conference on Reverse Engineering.*

Knowledge Discovery Metamodel (KDM) 1.0 Specifications, 2007: http://www.omg.org/cgi-bin/doc?ptc/2007-03-15

Larus J. R., 1999. Whole program paths. In *PLDI'99, Conference on Programming language design and implementation*.

Leduc J., 2004. Towards Reverse Engineering of UML Sequence Diagrams of Real-Time, Distributed Systems through Dynamic Analysis. Master's thesis, Carleton University.

Lethbridge T. C., 2003. The Dagstuhl Middle Model: An Overview. In *ATEM'03, 1st International Workshop on Metamodels and Schemas for Reverse Engineering.*

Müller H. A., Klashinsky K., 1988. Rigi – A System for Programming in-the-large. In *ICSE'88, 10th International Conference on Software Engineering*.

Nicola M., and John J., 2003. XML parsing: a threat to database performance", In *ICIKM'03, 12th*

International Conference on Information and Knowledge Management.

Reiss S. P. and Renieris M., 2001. Encoding program executions. In *ICSE'01, 23rd International Conference on Software Engineering*.

St-Denis G., Schauer R., and Keller R. K., 2000. Selecting a Model Interchange Format: The SPOOL Case Study. I*n Proc. of the 33rd Annual Hawaii International Conference on System Sciences*.

Systä T., 2000. Understanding the Behaviour of Java Programs. In *Proc. of the 7th Working Conference on Reverse Engineering (WCRE)*.

UML 2.0 Specifications, 2005: http://www.omg.org/technology/documents/formal/uml.htm

Woods S., Carrière S. J., Kazman R., 1999. A semantic foundation for architectural reengineering and interchange. In *ICSM'99, 15th International Conference on Software Maintenance.*

APPENDIX

In this appendix, we present the detailed semantics of CTF.

Class "Trace"

Semantics: An abstract class representing common information about traces generated from the execution of the system.

Attributes:

- startTime: Time - Specifies the starting time of the generation of the trace.
- endTime: Time - Specifies the ending time of the generation of the trace.

Associations: No associations.

Constraints:

- endTime should be greater than or equal to startTime
  self.endTime >= self.startTime

Class "CallTree (Subclass of Trace)"

Semantics: An object of CallTree represents a trace of routine calls. A routine is defined as any function whether it is in a class or not. Although the class refers to a tree but in fact it will be saved as an ordered DAG.

Attributes: No additional attributes.

Associations:

- root: Node[1] - Specifies the root of the call tree.

Constraints:

- The root of a trace must not have parent node
  self.root.incoming ->isEmpty()
- The root node cannot be an object of ControlNode subclasses:
  not self.root.oclIsTypeOf(SequenceNode) and
  not self.root.oclIsTypeOf(RecursionNode)
- The graph needs to be an ordered directed acyclic graph.

Class "TracePattern"

Semantics: An object of the class TracePattern represents a sequence of calls that is repeated in a non-contiguous manner in the trace.

Attributes:

- description: String - Specifies a textual description that a software engineer can assign to a trace pattern.

Constraints: No additional constraints.

Class "Node"

Semantics: Node is an abstract class that represents the nodes of the directed acyclic graph (i.e. compact form of the call tree).

Attributes:

- label: String - If a static component is not specified, perhaps because it is not known, then the label can be used to indicate the node's label. For example, a label can simply represent the name of the routine represented by this node.
- timestamps: Time[] - Specifies the timestamps of the routines represented by this node.
- executionTime: double[] - Specifies the execution times of the routines represented by this node.

Associations:

- DAG: CallTree [1] - References the Trace for which this node is the root.
- incoming: TraceEdge [*] - Specifies the TraceEdge objects that represent the incoming edges of this node.
- outgoing: TraceEdge [*] - Specifies the TraceEdge objects that represent the outgoing edges of this node.
- Thread [*] - References the Thread objects that represent the thread in which this node is executed.

Constraints:

- The timestamps of the routine calls represented by this node must be sorted in an ascending manner. This guarantees that the graph maintains the sequential execution of the routines.
- The parent nodes of this node cannot be the same as its child nodes and vice-versa since the graph is acyclic.
  self.incoming->excludesAll(self.outgoing) and

self.outgoing ->excludesAll(self.incoming)

Class "TraceEdge"

Semantics: Objects of the TraceEdge class represent edges of the directed acyclic graph.

Attributes:

- repetitions: int - Specifies an edge label that will be used to represent the number of repetitions due to loops and recursion. Default value is zero, i.e., no repetitions.

Associations:

- child: Node[1] - References the node that represents the child node that is pointed to by this trace edge.
- parent: Node [1] - References the node that represents the parent node from which this edge is an outgoing edge.

Constraints:

- The child and the parent nodes must be different. Recursion is represented using the RecursionNode class (Section 3.1):
  sefl.child <> self.parent
- The value of the attribute "repetitions" must be greater than or equal to zero
  self.repetitions >= 0

Class "Thread"

Semantics: Objects of the Thread class represent the thread of execution invoked in the trace.

Attributes:

- name: String - Specifies the name of the thread.

Associations:

- Node[1..*] - References the nodes that are executed in this thread of execution.

Constraints: No constraints

Class "RoutineCallNode (Subclass of Node)

Semantics: Objects of the RoutinceCallNode represent the routine calls invoked in the trace. A routine here should not be confused with a method of a class.

Attributes: No additional attributes

Associations: No additional associations.

Constraints: No additional constraints

Class "MethodCallNode (Subclass of Node)"

Semantics: Objects of the MethodCallNode represent the method calls invoked in the trace.

Attributes: No additional attributes.

Associations:

- Object [0..1] - References the object, if known, on which the method is invoked.

Constraints: No additional constraints

Class "Object"

Semantics: This class represents the objects invoked in the trace. In some traces, information about objects may be present; in others such information (and hence instances of this class) may be absent.

Attributes:

- objectID: String - Specifies the object identifier.

Associations:

- MethodCallNode [1..*]- Specifies the methods invoked on this object.

Constraints: No additional constraints

Class "ControlNode (Subclass of Node)"

Semantics: The ControlNode class is an abstract class that is used to specify additional information that can help better structure the trace.

Attributes: No additional attributes.

Associations: No additional associations.

Constraints:

- A control node cannot be the root of the entire trace:
  self.incoming ->notEmpty()
- A control node must have children:
  self.outgoing -> notEmpty()

Class "RecursionNode (Subclass of ControlNode)"

Semantics: An object of the RecursionNode is added to represent graph nodes that are repeated recursively. In this case, this object will be labeled 'REC'.

Attributes: No additional attributes.

Associations:

- repeatedOccurrence: Node[1] - References the subtree that is repeated recursively.

Constraints: No additional constraints

Class "SequenceNode (Subclass of Control Node)"

Semantics: An object of the SequenceNode class is added to represent multiple nodes that are repeated in a contiguous way. In this case, this object will be labeled 'SEQ'.

Attributes: No additional attributes.

Associations: No additional associations.

Constraints: No additional constraints.