STEP: A Framework for the Efficient Encoding of General Trace Data*

Rhodes Brown, Karel Driesen, David Eng, Laurie Hendren,
John Jorgensen, Clark Verbrugge and Qin Wang
Sable Research Group, McGill University
Montréal, Québec, CANADA H3A 2A7
{rhodesb,karel,flynn,hendren,jjorge1,clump,qwang21}@cs.mcgill.ca

ABSTRACT

Traditional tracing systems are often limited to recording a fixed set of basic program events. This limitation can frustrate an application or compiler developer who is trying to understand and characterize the complex behavior of software systems such as a Java program running on a Java Virtual Machine. In the past, many developers have resorted to specialized tracing systems that target a particular type of program event. This approach often results in an obscure and poorly documented encoding format which can limit the reuse and sharing of potentially valuable information. To address this problem, we present STEP, a system designed to provide profiler developers with a standard method for encoding general program trace data in a flexible and compact format. The system consists of a trace data definition language along with a compiler and an architecture that simplifies the client interface by encapsulating the details of encoding and interpretation.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Tracing*; D.3.4 [**Programming Languages**]: Processors—*Translator writing systems*; E.4 [**Data**]: Coding and Information Theory—*Data compaction and compression*

General Terms

Languages, Measurement, Performance, Standardization

Keywords

Data definition language, Program event trace, Sequential data encoding

*Further details and source code are available for download at: `http://www.sable.mcgill.ca/step/`

PASTE'02, November 18–19, 2002, Charleston, SC, USA

1. INTRODUCTION & MOTIVATION

Modern high-level languages such as Java provide a wealth of complex features such as type inheritance, integrated memory management (i.e., garbage collection), specialized control flow operations (e.g., virtual dispatch & exceptions) and language-level support for concurrent process control. These features, although intended to simplify the task of writing code, often complicate a number of other development issues such as software architecting, performance optimization and debugging. In recent years, a number of sophisticated static analyses have been developed to aide in program understanding and optimization. Blindly applied, these analyses often meet with varied success. We believe that many such analyses can be made more effective by incorporating an understanding of the run-time behavior of software systems. To this end, we have embarked on a number of projects to characterize the dynamic behavior of Java programs through the use of program traces. At the core of these efforts is STEP, a system designed for the efficient encoding of generalized program trace data.

Our approach is a departure from specialized trace formats such as PDATS [14], MaStA I/O [20], POSSE [11] and HATF [5]. The project was motivated by the observation that for complex software systems, such as Java programs running on top of a Java Virtual Machine, it is often unclear which program events should be recorded to obtain a "useful" characterization of the run-time behavior. Furthermore, such event traces may be generated in a variety of ways and may be used as input to a variety of tools and analyses. Complicating matters further is the fact that an accurate program characterization may require multiple large traces which, if stored in a naïve format, would place a strain on disk resources and limit sharing and reuse of the data. To address these issues, we established a set of basic requirements for a general trace encoding system:

- **Flexibility**: The system should provide a flexible trace format that is not bound to a particular set of data records. Specifically, adding new records to a trace should not break existing tools. Furthermore, the system should not be bound to any particular set of encoding strategies.

- **Integrated Documentation**: The trace files should be accompanied by a descriptive document that specifies both the form and [the] interpretation of the data records, including information related to encoding.

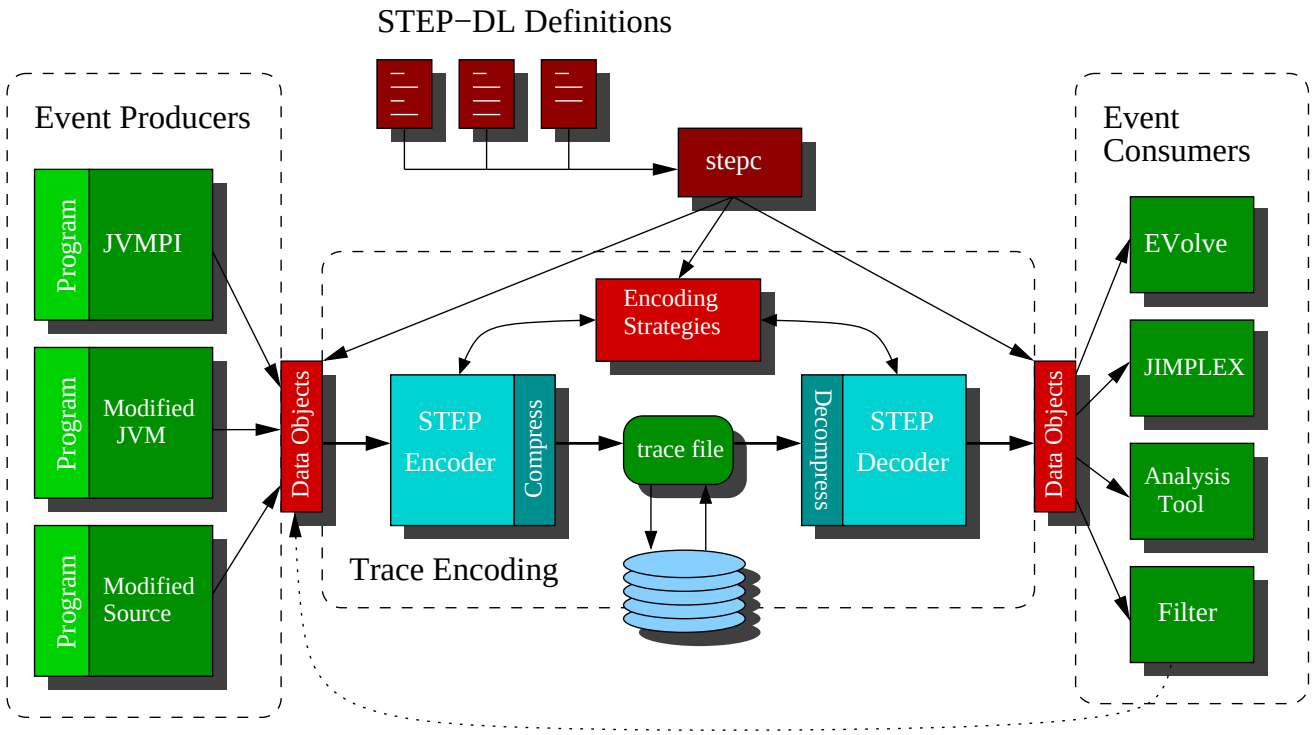

Figure 1: An overview of the STEP framework. This depiction shows the collection and analysis of Java program traces.

- **Compact Encoding**: The standard trace encoding should be based on an expectation that traces will be very large yet exhibit a high degree of sequential regularity. Strategies for trace reduction should be integral to the system, and consider both the average size of individual records as well as aggregate compression.[1] The default reduction scheme should be lossless, although lossy approaches should not be prohibited.
- **Encapsulation**: The basic client interface should be kept as simple as possible by isolating and encapsulating features such as encoding and other forms of translation.
- **Reuse**: The system should support reuse and extension of existing trace definitions and encoding strategies.

Based on these requirements, we set out to design a standardized yet flexible and compact trace encoding system. The result, depicted in Figure 1, has been dubbed STEP. The system acts as a bridge, connecting a variety of trace producers with a variety of trace consumers through the same uniform interface. Events are passed, as objects, from a producer to an encoding engine which serializes the data using a number of adaptive strategies to reduce the average event record size. Consumers use a symmetric decoding engine to reconstitute the event stream from a trace file. Clients do not implement the event definitions or encoding strategies directly. Instead, the definitions and strategies are specified with a definition language called STEP-DL. A compiler, `stepc`, reads the STEP-DL definitions and generates the necessary components for a particular target implementation (e.g., Java).

In the following sections we relate our experiences designing, implementing and using the STEP system. First, we focus on the design and features of our data definition language, STEP-DL. We then proceed to an overview of the encoding system and highlight some of the important design and implementation issues. We used STEP to collect trace data from Java programs using a variety of source agents and adapted several tools to analyze the resulting traces. In section 4, we relate some of our experiences using the system and briefly discuss the compression of some example traces. Section 5 presents a brief review of work related to this project. Finally, we provide some concluding remarks and discuss possible future work.

2. THE STEP DEFINITION LANGUAGE

The STEP system was inspired primarily by the work of Chilimbi, Jones & Zorn [5, 15] who developed a similar approach dubbed Meta-TF. It was clear to us that a specialized trace definition language, like Meta-TF, provided an effective way to satisfy our documentation criteria. However, our data sets were not particularly compatible with the initial version of Meta-TF. Additionally, we believed that a somewhat different choice of language features would support a solution that satisfied our other basic requirements. Thus, we began to develop a new approach which we referred to as the STEP Definition Language (STEP-DL).

We considered basing STEP traces on an existing mark-up approach, such as XML [26] or SGML [13], however such an

[1]To achieve the best overall compression, it is important to choose an approach that ensures a complementary interaction of these methods.

approach is inappropriate for two reasons: First, as noted by Chilimbi *et al.*, the verbose data tagging present in markup formats is incompatible with the key compactness requirement for traces. Second, the syntax for document type definitions (DTDs) in such languages is cumbersome and excessive for the task at hand. On the other hand, a specialized language can provide a concise, easily interpreted specification with an intuitive mapping from definitions to the data objects used by a client of the system.

Briefly recapping the development history of STEP-DL, the initial version resembled Meta-TF with a number of extensions. The language was then modified to include type inheritance, influenced initially by the DAFT language [10]. The current version of STEP-DL includes a distinct syntax with a number of important features:

- Records defined with STEP-DL are composed of a set of fields. Each field may be either a singular or array type. Initially, the records are defined in terms of basic field types such as `int`, `string` and `data`.[2] Once a record is defined, it may be used as a field type for subsequent records.
- A list of interpretation attributes, including encoding strategies, may be attached to any record or field structure. Attributes are grouped according to a particular function or application. For example, the current groups include "`encoding`", which specifies parameters for the default encoding format; "`map`", which indicates a method for converting one record into another (often useful when one event implies another); and "`property`", which indicates some inherent feature of the possible values.
- Record and field structures may be annotated with descriptive labels, intended for use with automated visualization and analysis tools.
- STEP-DL promotes the reuse of existing record definitions by providing an object-oriented style single inheritance mechanism. The key feature of this approach is the ability to inherit, extend or override the attribute values of fields inherited from a parent record.
- Since some users may choose to create their own particular definitions for common events (e.g., allocations, invokes, etc.), STEP-DL allows similar definitions to coexist through the use of packages.

To illustrate the features of STEP-DL, Figure 2 shows an excerpt from a definition hierarchy for JVMPI data. `JVMPI_Event` records are marked with the `property` attribute "event" to distinguish them as event records (vs. other, auxiliary record types). All JVMPI events have an `envId` field that indicates the thread in which the event occurred. The `encoding` attributes indicate that such values should be encoded with the "default" strategy (i.e., assume most values are the same, default value). The following line (currently commented out) indicates that such fields could also use the "identifier" strategy (see section 3.3) for multi-threaded applications.[3] `OBJECT_ALLOC` events are derived from the common

[2]The `data` type is used to contain arbitrary binary data.

[3]The `~` operator is used to extend a field's attributes. It is most commonly used to extend the attributes of inherited fields.

```
record JVMPI_Event
{
  <property:"event">

  int envId <property:"address">
            <encoding:"size=4,default">;

  // ~envId <encoding:"identifier">;
}

record OBJECT_ALLOC extends JVMPI_Event
{
  int arenaId <property:"address">
              <encoding:"size=4,identifier">;
    // -> there should be a limited (small)
    //    number of these

  int classId <property:"address">
              <encoding:"size=4,identifier">;
    // -> clearly an ID

  int isArray <property:"unsigned">
              <encoding:"size=1">;
    // -> T/F only needs 1 byte

  int size <property:"unsigned">;

  int objId <property:"address">
            <encoding:"size=4,delta=32768">;
    // -> assume allocated address is within
    //    +/-32768 of last
}
```

Figure 2: A STEP-DL definition

`JVMPI_Event` type, inheriting the `envId` field and "event" `property` attribute. The fields of an allocation record are attributed to indicate various encoding approaches. One notable approach is the "delta" strategy used for `objId` fields, which assumes that allocation addresses occur within 32768 of the last value and only transmits the difference (see Samples [19]).

2.1 Language Choices

Some of our design choices for STEP-DL warrant a brief explanation. We chose to allow inheritance of type definitions because it is a well established and familiar means of promoting reuse and extensibility. Our particular variant considers the inheritance of data and attributes (as opposed to data and functionality). Anticipating the wide variety of possible uses for attributes as "interpretive directives," our approach was to avoid a particular syntax for attribute values and instead separate the values into various groups where each group defines the exact syntax and semantics of the possible values. Users are free to add their own attribute groups, and we are currently investigating the addition of several tool-specific groups which could be used to further automate the integration with various trace consumers.

3. ENCODING ARCHITECTURE

Our definition language provides a means for satisfying the trace documentation requirement and a method for encouraging reuse. The architecture that accompanies STEP provides solutions for the compact encoding, encapsulation, and flexibility requirements.

```
StepRecordOutput stepOut = new StepEncodedOutput(file);
stepOut.write(new OBJECT_ALLOCATION(...));
```

(a) Producer

```
StepRecordInput stepIn = new StepEncodedInput(file);
StepRecord record = stepIn.read();
```

(b) Consumer

Figure 3: The basic STEP client interface

Figure 3 illustrates the basic client interface to a STEP stream. We present this rather trivial example to demonstrate the simplicity of the basic interface. Once the records have been defined, users need only concern themselves with the data in object format. Encoding details such as the byte ordering of integer values are completely hidden from the client.

3.1 The Encoding Process

One of the key differences between STEP and other trace encoding systems is its use of an adaptive encoding process. Instead of using a fixed encoding policy (or a dynamic policy with explicit changes, as is possible with Meta-TF), the system monitors various characteristics of the input data and, when appropriate, makes adjustments to the encoding policy automatically. The process is implemented by associating each record type with a separate encoder object. Each encoder encapsulates a policy for translating values of the given type to and from the binary representation. Some encoders implement a direct translation, while others implement a more sophisticated transformation based on properties of the underlying values. Encoders are arranged to form a tree- or DAG-like hierarchy, with record encoders deferring to sub-encoders to handle their various fields. The encoders (represented in figure 1 by the "Encoding Strategies") are assembled, as needed at run-time, by querying a definition object for the `encoding` attributes of a particular type. The definition objects are constructed from the STEP-DL definition for the given type. As records are received by the system, the encoders adjust their internal policy based on the parameters of their particular strategy, communicating their state changes in the form of *meta*-events. When the trace is decoded, the meta-events are applied so as to recreate the same sequence of policy adjustments made by the encoding process.

Encapsulating the encoding policies inside independent objects provides a great deal of flexibility. Encoders may be nested, chained or shared in a variety of ways and their interface makes few assumptions about the data being encoded. The design facilitates experimentation with encoding techniques, as new strategies can be added with only minor modifications. Furthermore, if the definition for a particular trace record is not available during the decoding process, the record is simply skipped with no effect on the other encoders.

3.2 Meta Records

The method of embedding policy changes directly in the data stream presents a unique challenge. Consider a record with an array field where one of the array values is irregular (thus requiring a meta-event). The meta-event cannot simply be inserted before the encoded record since the change to the decoder would be applied too early and the first several array elements would be incorrectly decoded. To address this problem, we use a special *meta*-record type, which contains of the change information (i.e., an encoded meta-event) along with partitioned record segments to be read before and after the change is reflected in the decoder's policy. Meta-records provide a fully general solution to the problem, allowing several meta-events to be applied at different points during the decoding of a record. Meta-records may even contain other meta-records. This is often useful as the meta-events are themselves implemented as STEP records and thus encoding them may generate meta-meta-events. The result is a potentially recursive encoding process Our implementation of the process attempts to address this requirement while preserving the efficiency of normal record encoding.

3.3 Identifier Data

One encoding strategy, in particular, has a significant effect on the compactness and compressibility of STEP traces, and warrants further discussion. The strategy arose from an early observation that our traces often contained fields that are limited to a certain fixed set of values. In our examples, the fields were strings which we termed *identifiers*. Clearly it is wasteful to store the full representation of such values (e.g. "`spec.benchmarks._213_javac.UnsignedShiftRightExpression`" for a `type` field) in each record. Instead, the identifier strategy encodes the values using a compact integer ID, signalling the mapping of value to ID with meta-events. The current version of STEP, extends this idea beyond string values to include arbitrary field data that exhibit an identifier distribution. For example, the `classId` field shown in Figure 2 encodes integer (address) values using the identifier strategy.

Experience has shown that a careful use of identifiers can significantly reduce the average record size. In fact, preliminary results indicate that the average field size can often be reduced to just over 1 byte—a nearly optimal byte-level encoding. This approach has important consequences with regards to aggregate compression, in that it allows more records to fit in a compressor's pattern space while preserving the sequential regularity of the record data.

3.4 Additional Design Factors

Like the approach of Haines *et al.* [10], the STEP system is designed to embody the major elements of Booch's [1] definition of an *object model*: abstraction (data objects appear uniform, while they exhibit variable encoding), encapsulation (clients are isolated from the encoding process), modularity and hierarchy (interpretation strategies are modular and composable; record types are extensible). The system also embodies some minor elements including typing, and persistence.

4. EXPERIMENTS & EXPERIENCE

In an effort to examine the utility of STEP, we considered three distinct tests of the system: 1) expressing a complex set of trace data with STEP-DL, 2) adapting several tools to read STEP traces and 3) evaluating the compressibility of the encoding format.

4.1 Using STEP-DL

To test the expressiveness of STEP-DL, we proceeded to define a reasonably complete set of events generated by a Java Virtual Machine. Our definition included common Java entities such as classes, methods and fields; VM entities such as bytecode addresses, and events such as allocations and invocations; and a complete hierarchy of bytecode events. We also defined a hierarchy for JVMPI event data, a portion of which is illustrated in Figure 2. The result was a successful exercise that produced over 300 record types and explored virtually every feature in the language.

4.2 Client Integration

One of the primary motivations in developing STEP was our need for a general interface to trace data that could be used by a variety of different trace generators, visualizers and analysis tools. We created STEP traces using several data sources, including JVMPI data, instrumented bytecode and an instrumented JVM. The resulting traces were used as input for two significantly different trace consumers: EVolve [27], an extensible tool for graphically analyzing event-based data, and JIMPLEX/JIL [6], a system for browsing intermediate Java code representations augmented with static and dynamic properties. Using the simple STEP interface, we were able to quickly and easily adapt these tools to read the same traces. In fact, the process of adapting the tools was so simple that it motivated our refactoring of STEP-DL attributes to support supplemental methods of interpretation and translation.

4.3 Trace Compression

We applied STEP to a series of allocation traces (including frees and GC events) obtained from a JVMPI profiling agent. The results are summarized in table 1. The traces were initially recorded in a "raw" binary format, using a single byte to mark the event type. The raw format used roughly 17 bytes per record (BPR) on average, whereas the STEP encoding often used 1/3 the number of bytes (with certain anomalies, believed to be the result of multi-threading). Furthermore the STEP traces were more compressible than the raw format using standard tools such as `gzip` [8] and `bzip2` [21]. Overall, the compressed STEP traces reduced the raw data to roughly 5% of its original size; and even better compression was possible when the traces were restricted to only allocation events. This appears to be a significant improvement over comparable results presented by Chilimbi *et al.*.

As Samples [19] points out in his work on address trace compression, the choice of record encoding strategy can have a dramatic effect on the overall reduction achieved by applying sequential compression algorithms such as those of Ziv & Lempel [28] and Burrows & Wheeler [4]. We attribute the compactness and compressibility of our heterogeneous, example traces to two factors: 1) The appropriate use of encoding strategies reduced the average field size to a nearly optimal byte-level encoding. 2) Secondary patterns in the STEP format (such as identifier values) resulted in a further skewing of the byte value distribution, thus artificially increasing the probability of certain byte sequences.

5. RELATED WORK

There is a large body of work devoted to the collection, compression and analysis of program trace data. We do not attempt to present a complete overview of this work.

Shende *et al.* developed TAU [22] for profiling C++ applications, and Reiss & Renieris [18] developed a system for profiling C, C++ and Java. While these systems bear resemblance to STEP and present several novel approaches to encoding, they focus primarily on invocation data and do not document their format or suggest how it might be extended.

The majority of work on program tracing relates to the collection and use of address traces. Uhlig and Mudge [25] survey much of this work and review a number of the lossy and lossless reduction techniques for address traces. The best lossless methods are based on variants of Samples' difference technique [19]. Fox & Grün [7] suggest a novel extension of this approach based on dynamic discovery of a program's structure, although it is unclear whether the technique could be generalized to other types of trace data.

A number of "standard" trace formats have been proposed; examples include: PDATS [14], MaStA I/O [20], POSSE [11] and HATF [5]. These formats each focus on a particular domain, and are not compatible with each other.

It is important to contrast STEP from program instrumentation systems such as ATOM [24] or EEL [16]. STEP is a trace encoding system. And although our experiments have focused on Java programs, the system does not make any assumptions about the data source, other than the regularity characteristic common to program traces. Also, STEP is not designed as a replacement for existing profile tools such as OptimizeIt [2], JProbe [23] or Jinsight [12]. Our goal is to provide a standard trace representation, so that a variety of tools can analyze the same trace, thus reducing development overhead and providing a means for meaningful comparison of results.

Tailored languages for the automated generation of file manipulation tools have been in use for many years (see, for example [17]). Our particular use of an object-oriented approach was inspired by the SmartFile system [10], used to encode scientific data files. Our approach differs, however, in that we focus on records as extensible objects instead of the entire file format. The Meta-TF system (first presented in [5], later refined in [15] and used to instantiate the HATF standard) bears a close resemblance to STEP and, in fact, was the primary inspiration for our overall approach. However, STEP differs from Meta-TF in its more general and extensible approach to data types and encoding strategies. Some of the particular differences include inheritance of record types, generalized identifier encoding, and extended support for interpretation attributes. Furthermore, by associating the encoding strategies with individual record types (as opposed to the system as a whole), the same interface may be used by all clients of the system—thus enabling a more general approach to visualization and analysis of arbitrary trace data.

6. SUMMARY

We have presented STEP, a system designed to facilitate the definition, encoding and sharing of arbitrary program trace data. The system was motivated by the need to capture the rich variety of events and behaviors exhibited by modern software systems such as Java programs running on a Java Virtual Machine.

The system includes a powerful data definition language, STEP-DL, which includes features such as type inheritance

Benchmark	Raw Size	Raw BPR	Raw.gz Size	STEP Size	STEP BPR	STEP.gz Size	% of raw size
sablecc	1083.42 MB	17.12	175.80 MB	417.11 MB	6.59	45.41 MB	4.19%
soot*	2048.00 MB	17.05	367.10 MB	1508.52 MB	12.56	108.80 MB	5.31%
compress	0.38 MB	19.64	0.08 MB	0.13 MB	7.05	0.03 MB	8.23%
jess	257.57 MB	17.00	46.70 MB	99.45 MB	6.56	12.21 MB	4.74%
db	104.27 MB	17.01	14.60 MB	36.78 MB	6.00	3.69 MB	3.54%
javac	205.55 MB	16.93	39.38 MB	79.20 MB	6.53	11.56 MB	5.62%
mpegaudio	0.49 MB	19.99	0.10 MB	0.17 MB	6.98	0.04 MB	7.22%
mtrt	215.55 MB	17.00	36.25 MB	226.16 MB	17.83	12.02 MB	5.58%
jack	194.01 MB	17.01	34.94 MB	68.62 MB	6.02	8.79 MB	4.53%

*Truncated at file size limit.

Table 1: Compression of STEP trace files.

and generalized attribute support. The `stepc` compiler uses the record and attribute definitions to generate a client interface with a set of encoding strategies. The system provides a complete encoding architecture, including a number of default encoding strategies.

The design of the system addresses a number of requirements including: a flexible and compact encoding format, integrated documentation, encapsulation of the encoding details and support for inheritance-based reuse. The features of the system build on a number of existing approaches to provide an effective and general solution.

We have tested the expressiveness of the definition language, the ease of integration with other tools, and the effectiveness of the default reduction strategies. Overall, we are pleased with the resulting utility of the system.

7. FUTURE WORK

We are currently exploring a number of uses and extensions of the STEP system. As mentioned in section 5, there is a wide range of literature regarding trace reduction strategies. Based on this work, we hope to expand the set of default STEP encoding strategies to provide a wider range of support for common data elements. Section 2 mentions that the `stepc` compiler may be extended to recognize various STEP-DL attributes. Using this mechanism, we are currently working on extensions to automate the generation of interface code for tools such as EVolve [27]. We also hope to develop automated filtering and augmenting tools, which either remove or supplement the records in a trace. Efforts are underway to develop other analysis tools that read STEP traces—including tools for statistical and pattern analysis. Also, we plan to use STEP as the basis for a comprehensive study of Java benchmark programs.

Acknowledgements

This work was funded by the Natural Sciences and Engineering Research Council of Canada (NSERC). Many thanks to members of McGill's Sable Research Group for their continued support of the project and instrumental help in establishing the objectives of the system, and designing the syntax and semantics of STEP-DL.

8. REFERENCES

[1] G. Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Object Technology Series. Addison-Wesley, Reading, MA, USA, 2nd edition, 1994.

[2] Borland Software Corp. OptimizeIt^{TM} suite. <`http://www.borland.com/optimizeit/`>.

[3] R. Brown, K. Driesen, D. Eng, L. Hendren, J. Jorgensen, C. Verbrugge, and Q. Wang. STOOP: The Sable toolkit for object-oriented profiling. Sable Technical Report 2001-2, Sable Research Group, McGill University, Montréal, QC, Canada, Nov. 2001.

[4] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital SRC Research, Palo Alto, CA, USA, May 1994.

[5] T. Chilimbi, R. Jones, and B. Zorn. Designing a trace format for heap allocation events. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pages 35–49, Minneapolis, MN, USA, Oct. 2000. ACM Press.

[6] D. Eng. Combining static and dynamic data in code visualization. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Anaylsis for Software Tools and Engineering (PASTE)*, Charleston, SC, USA, Nov. 2002. ACM Press.

[7] A. Fox and T. Grün. Compressing address trace data for cache simulations. Technical Report SFB 124, D4, Universität des Saarlandes, Saarbrücken, Germany, July 1996.

[8] J.-l. Gailly, M. Adler, and t. Free Software Foundation, Inc. The gzip (GNU zip) compression tool. <`http://www.gzip.org/`>.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, MA, USA, 1995.

[10] M. Haines, P. Mehrotra, and J. Van Rosendale. SmartFiles: An OO approach to data file interoperability. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 453–466, Austin, TX, USA, 1995. ACM Press.

[11] T. O. Humphries, A. W. Klauser, A. L. Wolf, and B. G. Zorn. The POSSE trace format version 1.0. Technical Report CU-CS-897-00, Department of Computer Science, University of Colorado, Boulder, CO, USA, Jan. 2000.

[12] IBM Research. Jinsight. <`http://www.research.ibm.com/jinsight/`>.

[13] Standard Generalized Markup Language (SGML). ISO Standard 8879, International Organization for Standardization, 1986.

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

[15] R. Jones. *Specifying trace formats: MetaTF 1.2.1.* Canterbury, Kent, UK, Mar. 2001.

[16] J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, La Jolla, CA, USA, 1995. ACM Press.

[17] L. M. Norton. A program generator package for management of data files–the input language. In *Proceedings of the ACM Annual Conference*, pages 217–222, Washington, DC, USA, 1978. ACM Press.

[18] S. P. Reiss and M. Renieris. Encoding program executions. In *Proceedings of the ACM SIGSOFT-SIGPLAN/IEEE Computer Society International Conference on Software Engineering (ICSE)*, pages 221–230, Toronto, ON, Canada, 2001. IEEE Computer Society Press.

[19] A. D. Samples. Mache: No-loss trace compaction. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 89–97, Oakland, CA, USA, 1989. ACM Press.

[20] 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.

[21] J. R. Seward. The bzip2 compression tool. <`http://sources.redhat.com/bzip2/`>.

[22] S. Shende, A. D. Malony, J. Cuny, P. Beckman, S. Karmesin, and K. Lindlan. Portable profiling and tracing for parallel, scientific applications using C++. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 134–145, Welches, OR, USA, 1998. ACM Press.

[23] Sitraka, Inc. JProbe. <`http://www.sitraka.com/software/jprobe/`>.

[24] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 196–205, Orlando, FL, USA, 1994. ACM Press.

[25] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys*, 29(2):128–170, June 1997.

[26] Extensible Markup Language (XML) 1.0. W3C recommendation, World Wide Web Consortium, 2000. <`http://www.w3.org/TR/REC-xml`>.

[27] Q. Wang, R. Brown, K. Driesen, L. Hendren, and C. Verbrugge. EVolve: An extensible software visualization framework. Sable Technical Report 2002-6, Sable Research Group, McGill University, Montréal, QC, Canada, June 2002.

[28] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–343, May 1977.

APPENDIX

A. STEP-DL 1.0 SYNTAX

EBNF

```
<def file> ::= <definition>*

<definition> ::=
  'package' <name> '{' <definition>* '}' |

  'record' <name> <label>?
    ( 'extends' '!'? <record name> )? '{'
    <description>?
    <attribute>*
    <field list>*
    <field modifier>*
  '}'

<attribute> ::= '<' <group> ':' <value> '>'

<field list> ::=
  <attributed type>
  <field def> ( ',' <field def> )* ';'

<attributed type> ::= <type> <attribute>*

<field def> ::= <name> <description>? <attribute>*

<field modifier> ::=
  ( '~' | '!' ) <field name> <attribute>+ ';'

<type> ::=
  'int'            |
  'string'         |
  'data'           |
  <user type name> |
  <attributed type> '[]'
```

Notes

- STEP-DL files are expressed in standard ASCII text.
- Type names may be relative to the current package or absolute.
- Attribute group names are standard identifiers. Attribute values are standard string literals.
- Field modifiers may be applied to any previously defined field (local or inherited). Modifier names may be qualified (using the `.` symbol) to refer to sub-fields (e.g., `x.length` refers to the length field of `string` types).
- In general, attributes are interpreted left to right and top to bottom (i.e., definitions first, then modifiers). The specific rules regarding which values dominate, or take precedence over others is deferred to the definition of the particular attribute group.
- Comment formats include '`//`' and '`#`' single line, and '`/* */`' multi-line variants.

B. STANDARD ENCODING ATTRIBUTES

The `encoding` attribute group is the most prominent and integral to the STEP system. The encoding techniques fall into 3 categories: general regularity strategies, which may be applied to any data value; specific regularity strategies, which target a particular data type; and simple, property based rules. The precedence of encoding strategies is based on these three categories. First, the most recent general strategy is applied. If an irregular value is encountered, the next available rule is used: either a targeted strategy or basic rule. Again, if a targeted strategy encounters an irregular value, it defers to the most recently defined basic rules. If no basic rule is given, the encoder factory assigns certain default rules. When a strategy must defer to its subordinate, the irregular value is indicated through the use of a meta-event record.

B.1 General Strategies

identifier
: This strategy is applied when the values are expected to derive from a relatively small, fixed distribution. As new values are encountered, they are written as $< value, ID >$ pairs. All subsequent occurrences of the value result in only the ID being written to the trace. The decoder reads the IDs and converts them to values based on the initial mapping.

constant
: This strategy assumes that all values for the given field are the same. The value is only written for the initial occurrence. If any subsequent value differs from the initial value, an error is generated.

default
: This strategy is similar the the constant strategy, but deviant values are allowed and are signalled with meta-data. This strategy is effective for fields which almost always have the same value.

B.2 Integer Strategies

Integer (`int`) field values may be encoded using a variety of targeted strategies and basic rules.

B.2.1 Targeted Strategies

delta=*threshold*
: This strategy assumes that values are generally within *+/-threshold* of the previous value, and only encodes the difference. This is a version of Samples' difference technique [19]. The strategy is useful for data such as allocation addresses where the values often exhibit a sequentially increasing pattern.

stride=*increment*
: This strategy assumes that values occur with a regular increment from the previous value. In such cases, nothing is written to the trace and the decoder reconstructs the value from the previous value and the increment.

offset=*base*
: This strategy assumes that values are clustered about a given base value and that it is more economical to transmit the difference from the base than the absolute value.

window=*threshold*
: This strategy can be viewed as an adaptive version of the offset strategy. The initial value is used as the base, and subsequent values are encoded as the offset from the initial value. If the difference exceeds the given threshold, the base is shifted.

B.2.2 Basic Rules

size=*fixed* | *start*.. | *min*+ | **creep**
: The number of bytes used for an integer value (i.e., its size) can be defined in a number of ways. The rule may state that values always use the same fixed number of bytes. The rule may begin using a particular size, and then grow to use more bytes as larger values are encountered. The resizing may be elastic, in the case where values requiring more than the minimum are rare. Finally, a variable size encoding may be used, where the high bit of each byte is used to signal whether more bytes should be read. The default rule is to use the variable size "creep" rule.

unsigned
: Values that are always ≥ 0 are indicated as unsigned. This rule is implied by the `property` attributes "unsigned" and "address", and is often omitted in favor of the `property` version.

B.3 String Rules

String (`string`) types currently have just a single basic rule which states character encoding of the string in bytes.

charset=UTF-8 | **US-ASCII** | **...**
: The encoding of `string` values parallels Java's string encoding rules. The default rule is to encode values using the UTF-8 character set.

B.4 Record Rules

type=variable | **default** | **constant**
: Since STEP supports inheritance of record types, it is possible that sub-types may be used in the place of a field's defined type. To avoid object slicing, the record encoder must indicate the type of the specific value. The strategy for tagging the type of a record value assumes that either a) the types are uniform, in which case the `default` or `constant` options are appropriate, or that b) a number of different types are used, in which case the `variable` option (based on the `identifier` strategy) is a better choice.

General Notes

- `string`, `data` and array objects write a length field when encoded. The length encoding strategy may be adjusted with a relative modifier (e.g., `~x.length <encoding:"default">`).
- The strategy for elements of an array field can also be changed by applying a modifier to the `element` field.