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

Sable Technical Report No. 2002-7

Rhodes Brown, Karel Driesen, David Eng, Laurie Hendren,
John Jorgensen, Clark Verbrugge and Qin Wang

15 June 2002

Contents

List of Figures

List of Tables

Abstract

Traditional tracing systems are often limited to recording a fixed set of basic program events. These limitations 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 architecture that simplifies the client interface by encapsulating the details of encoding and interpretation.

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 secondary tasks 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 have often met with varied success. We believe that a key factor in the effective application of program analyses lies in making informed choices based on 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.[1]

Our approach is a departure from specialized trace formats such as PDATS [13], POSSE [10] 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 a number of massive 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 interpretation of the data records, including information related to encoding.
- **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.[2] The default reduction scheme should be lossless—although, a lossy approach may be sufficiently accurate in some situations, and thus should not be prohibited.

[1]In this paper we refer to program *tracing* and program *profiling* interchangeably. Strictly speaking however, a program profile may be more general and include statistical and/or snapshot measurements, whereas a program trace refers specifically to a sequence of program events.

[2]In practice, the combination of methods for aggregate compression and record size reduction can have a significant effect on the overall reduction ratio. These methods can often interfere with each other in subtle ways. To achieve the best overall compression, care must be taken in choosing reduction methods that are complementary.

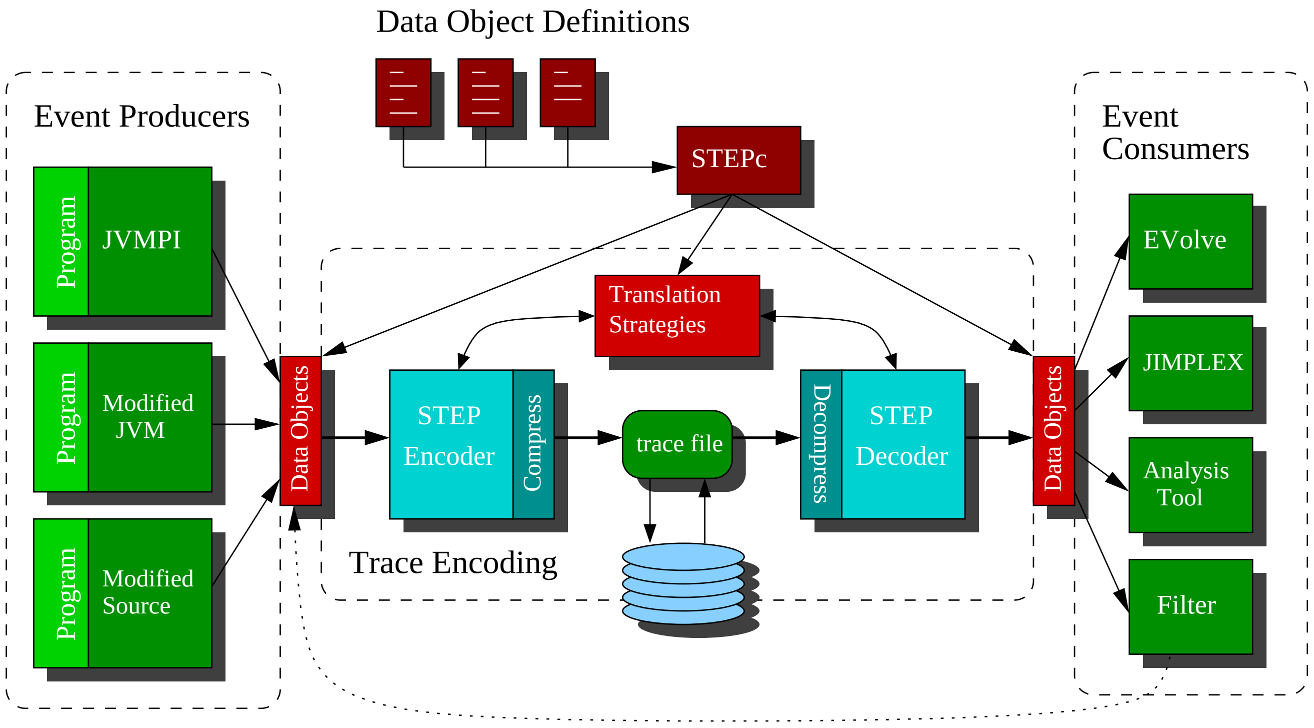

Figure 1: An overview of the STEP framework. This depiction of the system shows the collection and analysis of Java program traces. However, it is important to note that the system is designed to operate independently of any particular producer or consumer clients.

- **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 standardized yet flexible and compact trace encoding system. The result, depicted in Figure 1, has been dubbed STEP. The system provides an object-oriented data definition language, STEP-DL, a compiler for the language, `stepc`, and an encoding architecture which includes a number of default encoding strategies. We have used STEP to collect trace data for Java programs from a variety of sources and have adapted several tools to analyze the resulting traces.

In the following sections we relate our experiences designing, implementing and using the STEP system. First, we focus on the design and implications of our data definition language, STEP-DL. We then proceed to an overview of some of the major implementation issues and discuss some of the challenges we encountered while developing the system. Following this, we relate some of our experiences using the system and present some preliminary results regarding compression of some sample trace data. In section 5, we discuss some of the 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 *et al.* [5, 14] who developed a similar approach dubbed MetaTF. It was clear to us that a specialized trace definition language, like MetaTF, provided an effective way to satisfy our documentation criteria. However, our data sets were not particularly compatible with the initial version of MetaTF. Additionally, we believed that a somewhat different choice of language features would motivate a solution that satisfied our other basic requirements. Thus, we began to develop a new approach which we referred to a the STEP Definition Language (STEP-DL).

We considered basing STEP on an existing mark-up approach, such as XML [26] or SGML [12], however such an approach is inappropriate for two reasons: First, as noted by Chilimbi *et al.*, the verbose data tagging present in mark-up 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 MetaTF with a number of extensions. The language was then modified to include type inheritance, influenced initially by the DAFT language [9]. 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`.[3] 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 there own particular definitions for common events (e.g., allocations, invokes, etc.), STEP-DL allows similar definitions to coexist through the use of packages.[4]

To illustrate the features of STEP-DL, Figure 2 shows an excerpt from a definition hierarchy for Java bytecodes. The example illustrates how the `ifeq` and `invokevirtual` operations derive from the common `CompletedInstruction` event. Both events possess a `target` field that indicates the next bytecode to be executed, while `invokevirtual` also has an `object` field that specifies the target object. Inherited fields with modified attributes are indicated with the `~` operator. The `encoding` attributes indicate a strategy for encoding the values of a particular field. The "default" `encoding` attribute is reasonably straight forward, indicating that a value should only be included in the encoded record if it is different from the given default. The "identifier" encoding is explained in section 3. Other common attributes, such as `map` and `property` are also visible in the example.

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). Regarding the syntax and semantics of attributes, our particular choice was intended to accommodate the requirements of flexibility and encapsulation. The idea being to remove all interpretive issues from the

[3]The `data` type is useful as a container for arbitrary binary data.

[4]Although we implement STEP-DL packages as Java packages, the only intended semantics are those of name-space partitioning.

```
record ControlFlowChange extends CompletedInstruction {
  // Inherit: count, bytecode, pc, thread, popped, pushed
}

record TargetedControlFlowChange extends ControlFlowChange {
  PC  target <encoding:"identifier">; // next instruction
}

record ConditionalBranch extends TargetedControlFlowChange {
  ~popped <encoding:"default=1">; // pop the condition
  ~pushed <encoding:"default=0">; // push nothing
}

record ifeq extends ConditionalBranch {
  ~bytecode <encoding:"default=Bytecode(\"ifeq\",153)">;
}

record InvokeOp extends TargetedControlFlowChange {
  <map:"Invoke(bytecode, target.method, pc)">
}

record ContextInvoke extends InvokeOp {
  int  object <encoding:"length/width=4"><property:"address">;
}

record invokevirtual extends ContextInvoke {
  ~bytecode <encoding:"default=Bytecode(\"invokevirtual\",182)">;
}
```

Figure 2: A STEP-DL definition

```
StepFileOutput stepOutput = new StepFileOutput(fileName);
stepOutput.write(new Allocation(bytecode, type, size, pc));
```

(a) Producer

```
StepFileInput stepInput = new StepFileInput(fileName);
StepObject o = stepInput.read();
```

(b) Consumer

Figure 3: The basic STEP client interface

primary client interface, and instead create extensible secondary interpretation interface. Users are free to add their own attribute groups, and we are currently investigating modular extensions to the `stepc` compiler that are able to read such attributes and automatically generate an interface for various tools.

3 Implementation

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.

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, the encoding process is completely encapsulated. Issues such as byte ordering (endian) are hidden from the client.

3.1 Translators

The basic approach to encoding is to associate each record and field definition with a particular translation strategy. The strategy dynamically adjusts its encoding scheme as various values are encountered. A common example is integer fields where the values regularily require only a single byte to encode, but occasionally must be resized to fit larger values.

Translators are arranged in a tree- or DAG-like hierarchy, with record translators deferring to sub-translators to encode their various fields. When a translator changes its encoding scheme, a message is added to the trace indicating the change. Due to space limitations, we do not detail the translation process here, however it is worth noting that the process involves more than the trivial delegation suggested above. In short, the translation changes are reflected by writing *meta*-records to the trace, which consist of the change information along with partial results to be decoded before and after the change is applied to the decoding translator.

The encapsulation of the translation process inherently provides a great deal of flexibility. Translators may be nested, chained or shared in a variety of ways and their interface makes few assumptions about the data being encoded. If a translator is not available to read a particular trace record, the record is simply skipped, with no effect on the client consumer. Clients may chose to use the translator hierarchy generated from a STEP-DL specification, or assemble their own custom encoding strategies.

3.2 Identifier Data

One translation 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 through meta-records. The current version of STEP, extends this idea beyond string values to include arbitrary field data that exhibit an identifier distribution. For example, the `target` field shown in Figure 2 encodes a `PC` value using the identifier strategy. Chilimbi *et al.* suggest the possibility of using string tables to encode such fields, but do not consider more general values.

Preliminary results indicate that an careful use of identifiers can often reduce the average field size to just over 1 byte—a nearly optimal byte-level encoding. Furthermore, a consequence of using identifiers is a reduction in the average record encoding size, which allows more records to fit in a compressor's pattern space, thus enabling better compression. Of particular importance is that identifiers lead to a compact format while leaving sequential record patterns unchanged.

3.3 Additional Design Factors

Like the approach of Haines *et al.* [9], 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 (simple clients are isolated from the translation process), modularity and hierarchy (interpretation strategies are modular and composable; trace 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 (partially illustrated in Figure 2). The result was a successful exercise that produced nearly 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

Using the SOOT [25] bytecode transformation tool, we instrumented a number of the SPECjvm98 [23] benchmark programs to collect a trace of method entries and exits, allocations and field accesses. The traces made regular use of the identifier paradigm to encode values such as class, method, field and thread names.

Benchmark	Trace Size	`gzip` Size	`bzip2` Size
_202_jess	136.17 MB	1237.55 KB	247.11 KB
_213_javac	42.75 MB	1180.74 KB	521.06 KB
_222_mpegaudio	70.73 MB	825.69 KB	313.87 KB
_228_jack	53.13 MB	670.91 KB	168.49 KB

Table 1: Compression of STEP trace files

Table 1 shows that the STEP format is highly compressible using standard compression tools such as `gzip` [7] and `bzip2` [19]. For example, a trace from `jess` was reduced to only 0.18% of its original size using `bzip2`.

As Samples [18] 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 remarkable compactness and compressibility of our heterogeneous, example traces to two factors: 1) The appropriate use of identifiers reduced the average record size to roughly 5 bytes, a nearly optimal byte-level encoding. 2) Maintaining separate identifier mappings for each field has 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 [20] for profiling C++ applications, and Reiss & Renieris [17] developed a system for profiling C, C++ and Java. While these systems bare 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 [24] survey much of this work and review an number of the lossy and lossless reduction techniques for address traces.

A number of "standard" trace formats have been proposed; examples include: PDATS [13], POSSE [10] 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 [22] and EEL [15]. 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 [21] or Jinsight [11]. 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 [16]). Our particular use of an object-oriented approach was inspired by the SmartFile system [9], 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 MetaTF system (first presented in [5], later refined in [14] and used to instantiate the HATF standard) bares a close resemblance to STEP and, in fact, was the primary inspiration for our overall approach. However, STEP differs from MetaTF 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, the technique of associating an encoding strategy with each record type (as opposed to the system as a whole) allows the same interface to be used by all clients of the system—thus enabling a more general approach to visualization and analysis of arbitrary trace data.

6 Conclusions

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 and generalized attribute support. The `stepc` compiler uses the record and attribute definitions to automatically generate a client interface and 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 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.

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. Sable Technical Report 2002-4, Sable Research Group, McGill University, Montréal, QC, Canada, June 2002.

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

[8] 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.

[9] 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.

[10] 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.

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

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

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

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

[22] 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.

[23] Standard Performance Evaluation Corp. SPECjvm98 Java benchmarks. <`http://www.spec.org/osg/jvm98/`>.

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

[25] R. Vallée-Rai and the Sable Research Group. SOOT: a Java optimization framework. <`http://www.sable.mcgill.ca/soot/`>.

[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.

A STEP-DL 1.0 Syntax

EBNF

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

<definition> ::=
  package <name> ‘{’ <definition>* ‘}’ |

  record  <name> <label>? ( extends <record name> )?
  ‘{’ <description>? <attribute>* <field definition>* ‘}’

<attribute> ::= ‘<’ <key> ‘:’ <value> ‘>’

<field definition> ::=
  <attributed type> <field> ( ‘,’ <field> )* ‘;’ |  // new
  ( ‘~’ | ‘!’ ) <field name> <attribute>+ ‘;’       // extended

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

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

<type> ::=
  int | string | data | <user type name> |         // singular
  <attributed type> ‘[’ ( <int const> | ‘*’ ) ‘]’  // array
```

Notes

- Comment formats include ‘//’ and ‘#’ single line, and ‘/* */’ multi-line variants.