McGill University

STOOP: The Sable Toolkit for Object−Oriented Profiling

General Explanation

The performance and behavior of object−oriented programs is often very difficult to understand, particularly for large, complex software systems consisting of many packages and classes. STOOP provides high−level facilities which allow a user to rapidly construct tools to collect and visualize profile data from the execution of object−oriented programs.

As illustrated in the Figure to the right, there are three main components in the STOOP framework: a profiling agent, an event pipe, and a visualizer.

The profiling agent collects various profile data and uses it to generate a stream of events. Our intent is to support a wide variety of profiling agents including agents using the Java Virtual Machine Profiling Interface (JVMPI), instrumented virtual machines, and instrumented bytecode. It is also possible to use previously−recorded traces (off−line input).

The event pipe serves as a high−level interface between the front−end profiling agent and the back−end visualizer. It converts profile data to a binary format, buffers (and potentially compresses) the data, then forwards it to the visualizer. In order to provide a clear and flexible interface, profile events are described using the STOOP Trace Event Protocol: STEP. A compiler, STEPc, generates code that converts high−level representations of the profile data to a form that can easily be written to, and read from the event pipe.

The visualizer reads the stream of profile events leaving the pipe and presents the data using several different views. These views may take different forms according to the desired goals. We show some example views on the right.

The STEP system was inspired by the Meta−TF language introduced by Chilimbi, Jones & Zorn (ISMM '00) and by the SmartFile system introduced by Haines, Mehrotra & Van Rosendale (OOPSLA '95). Since our primary goal was to create a framework that could accept arbitrary profile data, the notion of using a profile specification language was naturally appealing. Originally, we had hoped to use the Meta−TF language directly, however we found it useful to modify and recast the ideas in a format that resembles the SmartFile definition language. The result is a distinct STEP definition language.

Our approach is unlike commercial profilers such as OptimizeIt, JProbe and Jinsight in that these systems are fixed profiling tools where the user interacts with the profiler to view predetermined program properties. The purpose of our tool is to allow one to develop profilers to view both standard and non−standard program properties, and to view events that come from a variety of sources.

STOOP has been implemented in Java and we have applied the toolkit to a variety of tasks for profiling Java. We are particularly interested in applying the toolkit to study program behavior that might suggest new optimization or execution strategies for Java. For example, by examining the behavior of hot data fields and the relationships between field accesses we may develop new strategies to make better use of a data cache.

Credits

Rhodes Brown — Karel Driesen
John Jorgensen — Laurie J. Hendren
Qin Wang — Clark Verbrugge

School of Computer Science
McGill University

This research supported by NSERC

Thanks to:
Transvirtual for the Kaffe virtual machine (www.kaffe.org).
Our predecessors in the Sable group for the Soot Java Optimization Framework (www.sable.mcgill.ca/soot).
Etienne M. Gagnon for assistance in printing the poster

Profiling Agent: Simple Example

```
/** A simple Java profile agent.
 *  The program being profiled indicates various
 *  events by invoking the corresponding method
 *  of the singleton Collector. Calls to the
 *  Collector may be inserted by hand or by a
 *  tool such as Soot.
 */
public class Collector {
  private static Collector _theInstance;
  private StepOutputStream _stepOut;

  private Collector(OutputStream os) {
    _stepOut = new StepOutputStream(os);

    Runtime.getRuntime().addShutdownHook(
      new Thread() {
        public void run() {
          _stepOut.flush();
          _stepOut.close();
        }
      });
  }

  public void recordClassLoad(String className,
                              String method) {
    _stepOut.writeRecord(
      new ClassLoad(
        new StringField(className),
        new IdField(method)));
  }

  public void recordMethodEnter(String method) {
    _stepOut.writeRecord(
      new MethodEnter(new IdField(method)));
  }

  public void recordMethodExit(String method,
                               String exitSite) {
    _stepOut.writeRecord(
      new MethodExit(
        new IdField(method),
        new IdField(exitSite)));
  }

  // other event handlers...

  // access the singleton Collector
  public static Collector v() { return _theInstance; }

  public static void main(String[] args) {
    _theInstance =
      new Collector(new FileOutputStream(args[0]));

    // run code to be profiled
  }
```

Profiling Agent: JVMPI Example

```
/** A JVMPI Event handler.
 *  Here we receive events from the JVMPI framework
 *  and process them by calling wrapper routines
 *  (Report* calls) that talk to the event pipe
 *  interface.
 */
static void NotifyEvent(JVMPI_Event *event) {
  JNIEnv *env = event->env_id;

  switch (event->event_type) {
    case JVMPI_EVENT_CLASS_LOAD: {
      ReportClassLoad(event->u.class_load.class_name,
                      CurrentMethod(env));
    }
    break;

    case JVMPI_EVENT_METHOD_ENTRY: {
      ReportMethodEntry(env, event->u.method.method_id);
    }
    break;

    case JVMPI_EVENT_METHOD_EXIT: {
      ReportMethodExit(env, event->u.method.method_id);
    }
    break;

    /* other event handlers... */
  }
}
```

Other profiling agents may include:

− Instrumented JVMs
− Off−line static traces

The STEP Definition Language

The STEP Definition Language and its associated STEPc compiler, allow users to quickly and easily add new record types to their profiling system.

The three main constructs in a STEP definition file are: packages, sessions and records. Of these, only records are required. A package definition is simply a namespace partitioning, similar to Java packages. A STEP session is used to define contextual information that is shared among several record types.

A STEP record is defined as a collection of fields. There are several basic field types, namely: int, string, identifier and data. Identifier fields are a convenient extension of strings that allow common values to be represented more compactly in the trace file. Data fields are provided as containers for arbitrary binary data. Once a record has been defined in terms of the basic types, it may in turn be used as a field type in subsequent records. Also, a field may be defined as an array of simple field types.

example1.step:

```
/* Example STEP record definitions */
package step {
  package example {
    record ClassLoad {
      className : string;
      method    : identifier;
    }

    record FieldAccess {
      declaringClass : identifier;
      fieldName      : identifier;
      readWrite      : int;
      // use an int as a boolean
    }

    record MethodEnter {
      method : identifier;
    }

    record MethodExit {
      method   : identifier;
      exitSite : identifier;
    }
  }
}
```

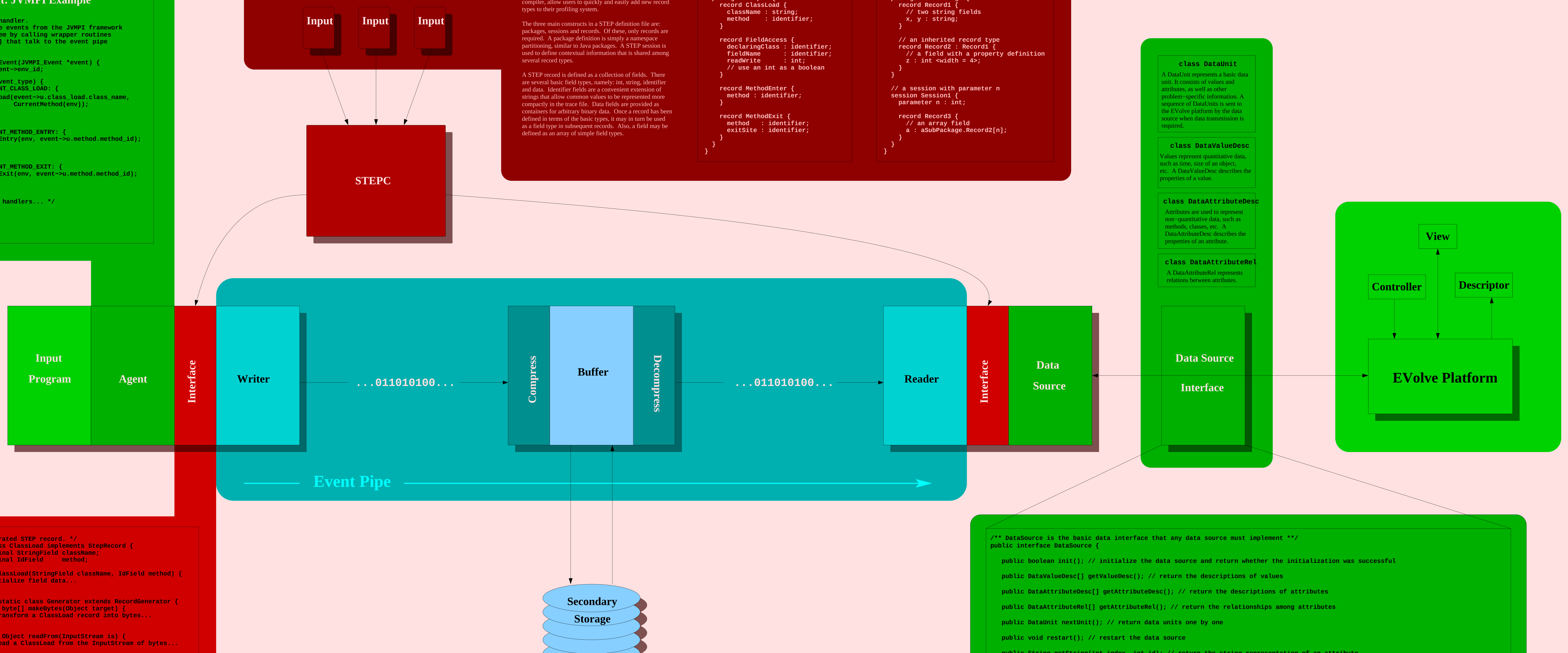
example2.step:

```
/* Some examples of the STEP−DL syntax */
package _a_Package_1 {
  package aSubPackage {
    record Record1 {
      // two string fields
      x, y : string;
    }

    // an inherited record type
    record Record2 : Record1 {
      // a field with a property definition
      z : int <width = 4>;
    }
  }

  // a session with parameter n
  session Session1 {
    parameter n : int;

    record Record3 {
      // an array field
      a : aSubPackage.Record2[n];
    }
  }
}
```



class DataUnit

A DataUnit represents a basic data unit. It consists of values and attributes, as well as other problem−specific information. A sequence of DataUnits is sent to the EVolve platform by the data source when data transmission is required.

class DataValueDesc

Values represent quantitative data, such as time, size of an object, etc. A DataValueDesc describes the properties of a value.

class DataAttributeDesc

Attributes are used to represent non−quantitative data, such as methods, classes, etc. A DataAttributeDesc describes the properties of an attribute.

class DataAttributeRel

A DataAttributeRel represents relations between attributes.

The STEP Interface

The STEP interface provides a simple abstraction for manipulating trace data as records, as well as methods for reading and writing the records in a compact binary format.

A STEP record is simply an aggregation of various fields. In this particular example, the ClassLoad record has two simple fields: a string and an identifier. In general, a wide variety of field types are available (see the '"STEP Definition Language" section above).

Each record type has an associated generator. A particular generator instance contains the information necessary to convert a record to and from the STEP binary format.

```
/** A generated STEP record. */
public class ClassLoad implements StepRecord {
  public final StringField className;
  public final IdField     method;

  public ClassLoad(StringField className, IdField method) {
    // initialize field data...
  }

  private static class Generator extends RecordGenerator {
    public byte[] makeBytes(Object target) {
      // transform a ClassLoad record into bytes...
    }

    public Object readFrom(InputStream is) {
      // read a ClassLoad from the InputStream of bytes...
    }
  }

  public Generator newGenerator(StepSession session) {
    return new Generator( ... );
  }
```

```
/** DataSource is the basic data interface that any data source must implement **/
public interface DataSource {

   public boolean init(); // initialize the data source and return whether the initialization was successful

   public DataValueDesc[] getValueDesc(); // return the descriptions of values

   public DataAttributeDesc[] getAttributeDesc(); // return the descriptions of attributes

   public DataAttributeRel[] getAttributeRel(); // return the relationships among attributes

   public DataUnit nextUnit(); // return data units one by one

   public void restart(); // restart the data source

   public String getString(int index, int id); // return the string representation of an attribute

   public int getTarget(DataAttributeRel relation, int id); // return the target of an attribute according to the given relationship
}
```

Sable Research Group

www.sable.mcgill.ca

Profiling Agent (left side of diagram)

Profiling agents provide raw data to the rest of the STOOP framework. Data may originate from a wide variety of sources. For example, traces of method invocations or object instantiations might come from a agent communicating with a virtual machine via the Java Virtual Machine Profiling Interface (JVMPI), maps of object addresses in memory can be produced by an instrumented garbage collector, and arbitrary application−specific data might come from custom instrumentation compiled into the profiled program.

To avoid restricting the set of possible data sources, STOOP imposes minimal requirements on profiling agents. Any information source−−−including existing trace data and sources as yet unforeseen−−−may feed a STOOP profiling agent, so long as the data it produces can be described in the specification language for STEP, the STOOP Trace Event Protocol. The specification of the data stream is compiled into a set of interface methods which the profiling agent calls to send its data to other components of the STOOP framework.

Profiling agents created to date include:

− a JVMPI agent which reports class loading and method entry/exit;
− an instrumented copy of the Kaffe virtual machine (www.kaffe.org), which reports field accesses as well as class loading and method entry/exit;
− instrumented Java class files produced from existing programs using Soot (www.sable.mcgill.ca/soot), a framework for analyzing and transforming Java bytecode. The transformed classes report their own field accesses as well as method entries and exits.

Description of Event Pipe (center of diagram)

The STOOP event pipe and the associated STEP system serve two purposes. First, the STEP system provides a simple, intuitive interface for manipulating trace data. The data is abstracted into simple records that can be accessed similar to C structs. Each record has an associated generator that knows how to convert the record into a compact, platform independent binary format. This encoding process is the second purpose of the event pipe.

Once a profiling agent has been created to collect various trace data, it is a relatively simple process to transform the data into the STEP binary format. First, trace records are defined using the STEP Definition Language (see panel above). The definitions are then fed to the STEPc compiler which creates Java or C code that is added to the profile agent. Both the profiling agent and the visualization DataSource can then manipulate the data using a high−level representation and neither need be concerned with the details of packing and unpacking the trace records. The generated STEP interface totally encapsulates the encoding.

Although the data abstraction provided by the STEP interface is convenient, the true benefit of the STEP system is the binary encoding of the trace data. In many cases, profiling agents may collect millions of trace events. However, it is often the case that the stream of events is highly 0. The STEP binary encoding is designed to facilitate the compression of such streams. Preliminary results indicate that some traces can be compressed to 0.2% of their raw size; a significant finding if one is considering raw traces in excess of a gigabyte.

Description of Visualizer (right side of diagram)

EVolve is the visualization part of STOOP. It is an extensible environment that simplifies the creation of new visualizations, and focuses on providing maximum extensibility and flexibility. The functionality of EVolve is primarily achieved by the combination of several components, which are supported by the EVolve platform. EVolve provides several built−in components, and new components can be created through inheritance.

The EVolve platform is the kernel of the whole environment; it takes charge of organizing the components to make them work as a single entity. Furthermore, the platform handles the communication among components, and also works as the connection between the data source and the components.

Components are classified into three types: views, controllers, and descriptors. The views are the core of the visualization. They construct and display graphs as well as interact with the user. A view receives messages from controllers or other views and can send messages to the descriptors.

Controllers receive command from the user and tell the views how to construct the graph accordingly. Unlike a view, a controller only sends out messages but doesn't respond to messages sent by other components. Examples of controllers are filter and palette.

When the users find what they are interested in from the view and need detail information about it, the view asks the descriptor to show it. A descriptor is exactly the opposite of a controller, in that it only receives messages from other components without sending messages out. A source code reader is one of the general−purpose descriptors in EVolve.