WG1.07.044

PSF Standardisation Initiative

WP1.4 Meeting
6 Nov 2007, Harsum (Tele Atlas)

DataScript Instance Parsing and Data Model Extensions

Harald Wellmann, Harman/Becker

Overview

- **Current status**
- **Points for improvement**
- **Solution Approach**
- **Usage Examples**
- **Proof of Concept Demo**

Current Status

- **A data model is specified in a set of *.ds files**
- **rds Compiler**
  - parses the DataScript source files
  - builds an Abstract Syntax Tree
  - traverses the tree to generate target language code
  - supported target languages: Java, C++, (HTML, XML)
- **Generated code:**
  - one class per DataScript type with methods `read()`, `write()`, `equals()`
  - some global classes (e.g. `__Visitor`) with one method per DataScript type
  - generated classes depend on a small run-time library
- **Decoding DataScript instances:**
  - constructor or `read()` method reads the input stream and builds a tree of Java/C++ objects reflecting the DataScript type hierarchy
  - application code traverses the object tree

Areas for Improvement

- **The DataScript object tree is just an intermediate product. An application needs to transform members of the tree and copy the content into its own structures.**
- **The tree is fairly expensive to build.**
- **The amount of generated code is very large.**
  - Java code from PSI trunk: ~450 classes, ~75 000 Lines of Code
- **There is no support for *forward compatibility*:**
  - **"Forward compatibility** is the ability of a system to accept input intended for later versions of itself."
  - Assuming model version 1.0 is a subset of model version 1.1, an application built for model version 1.0 shall be able to work with data stream instances complying to model version 1.1

Ideas

- **Use an event-based instance parsing approach, similar to SAX XML parsers.**
- **Use permanent IDs on DataScript types and fields.**
- **Add a compact binary representation of the DataScript model to the DataScript instance (= PSI database).**
  - i.e. the database describes its own format
- **Generate a list of target language constants for permanent IDs**
  - Just constants, no logic.
  - This is the *only* generated code required for the application.
- **The DataScript runtime library contains a generic instance parser.**
  - The parser first loads the binary model.
  - Using this model representation, the parser knows how to parse an instance of given type
  - The parser fires events with a permanent type or field ID for every parsed member.
- **Forward compatibility:**
  - The application simply ignores all events with unknown permanent IDs.

Analogy between DataScript and XML

	DataScript	XML
Model Source	*.ds files	*.xsd file (XML Schema Definition)
Binding of Model to Java Target Language	Java Code generated by rds	JAXB
Instance of Model	bitstream	*.xml file
Tree-based Instance representation	instances of rds-generated classes	DOM
Event-based instance representation	new: EDSI (Event-based DataScript Instance Parsing)	SAX

Instance Parser Interface

- The application obtains a `DataScriptInstanceParser` from a factory object.
- The application sets an instance handler to receive parser events.
- To parse an instance of a given type, invoke `parse(typeId)` with the type id of that type.

```
public interface DataScriptInstanceParser
{
   public void setInstanceHandler(DataScriptInstanceHandler handler);
   public void parse(int typeId) throws IOException;
}
```

Instance Handler Interface

- `DataScriptInstanceHandler` is the equivalent of a SAX Content Handler
- `startInstance()`, `endInstance()` indicate the start and end of the parsing process.
- Whenever entering or leaving a non-atomic field, the parser fires a `start...` or `end...` event.
- For atomic fields, the parser fires an event containing the field value.

```
public interface DataScriptInstanceHandler
{
    public void startInstance(int typeId);
    public void endInstance(int typeId);

    public void startCompound(int fieldId);
    public void endCompound(int fieldId);

    public void startArray(int fieldId);
    public void endArray(int fieldId);

    public void startArrayElement(int fieldId);
    public void endArrayElement(int fieldId);

    public void integerField(int fieldId, long value);
    public void bigIntegerField(int fieldId, BigInteger value);

    public void stringField(int fieldId, String value);
    public void enumField(int fieldId, int value);
}
```

DataScript Binary Data Model

- **Use DataScript to define a metamodel of itself.**
- **The rds parser builds an Abstract Syntax Tree (AST) of a DataScript model.**
- **This AST can be serialized to a compact binary form.**
- **Thus, for decoding the model of a DataScript instance, all we need is a DataScript instance parser.**

```
Model {
    int32        numTypes;
    Type         types[numTypes];
    int32        numFields;
    Field        fields[numFields];
    int32        numExpr;
    Expression   expressions[numExpr];
    int32        numPackages;
    Package      packages[numPackages];
    int32        numNames;
    string       names[numNames];
};

Field {
    int16    pos;
    NameId   name;
    TypeRef  type;
    bit:1    isOptional;
    bit:1    hasConstraint;
    bit:1    hasAlignment;
    bit:1    hasLabel;
align(8):
    ExpressionId  optional if isOptional;
    ExpressionId  constraint if hasConstraint;
    int32         alignment if hasAlignment;
    ExpressionId  label if hasLabel;
};
```

Forward compatibility

- **To achieve forward compatibility, type and field IDs used in the binary DataScript model are required to be invariant.**
- **The DataScript compiler shall support incremental compilation**
  - reading the current version of the model in textual form,
  - reading the former version of the model in binary form
  - and compiling a binary form of the current model, making sure that all types keep their IDs and that new types receive new IDs.
- **This will only work if the model is extended by inserting new elements or values.**
- **Reordering or deleting fields is not allowed.**

Parsing a future format instance

- When an older application version parses an instance of a newer format version, it will receive events with unknown IDs for any format extension.
- For atomic types, the application simply ignores the event.
- For complex types, the application ignores all events between the start and end events of the given compound type.
- This can be optimized by implementing a `setEventsEnabled(boolean)` method in the parser.

Implementation Issues

- **A proof-of-concept prototype of an EDSI parser in Java is available.**
  - This parser does not (yet) work with a binary data model.
  - It uses rds to parse the DataScript source and works with the AST of rds.
- **A full reference implementation shall be provided in Java.**
  - Useful for speeding up the viewers and diagnosis tools.
- **The EDSI design can easily be translated to C++.**
- **In C, the `DataScriptInstanceHandler` Interface can be emulated by a structure of function pointers.**

Extensions to rds

- **DataScript metamodel**
- **Incremental compilation of binary models**
- **Code generation for permanent IDs in multiple target languages and idioms.**

- **The run-time library implementation for a given target language only depends on the binary DataScript metamodel.**
- **EDSI run-time libraries can be developed independent of rds.**