The Serialization Killer Language

Timm Felden

May 29, 2013

Abstract

This work presents an alternative to various serialization approaches. The proposed serialization mechanism is fast, robust, extensible and easy to use. These goals are achieved by not using a human readable serialized form. To do (1)

Acknowledgements

Main criticis: Erhard Plödereder and Martin Wittiger.
Additional criticis: Dominik Bruhn.

To do (2)

To do (3)

To do (4)

Contents

1 **Motivation** 3
1.1 Related Work 3

2 **Syntax** 5
2.1 The Grammar 5
2.2 Reserved Words 6
2.3 Examples 6

3 **Semantics** 8
3.1 Includes 8
3.2 `annotation` 8
3.3 Sub Types 8
3.4 `const` 8
3.5 `auto` 9
3.6 Abstract Data Types 9
3.7 Comments 9

4 **The Type System** 9
4.1 Legal Types 10
4.2 Type Order 10
4.3 Strings 11
4.4 Compound Types 11
4.5 NULL Pointer 11
4.6 Examples 11

5 **Type Annotations** 13
5.1 Restrictions 13
5.2 Hints 16

6 **Serialization** **18**
6.1 Steps of the Serialization Process 18
6.2 General File Layout 18
6.3 Storage Pools 19
6.4 Pool Elements 19
6.5 Endianness 20

7 **Deserialization** **21**

8 **In Memory Representation** **22**
8.1 API 22
8.2 Representation of Objects 23
8.2.1 Proposed Data Structures 24
8.2.2 Reading Unmodified Data 24
8.2.3 Modifying Data 25
8.2.4 Reading Data in an Modified State 25
8.2.5 Writing Data 25
8.2.6 Final Thoughts on Runtime Complexity 25

9 **Case Study: Skill Encoded XML** **27**

10 **Future Work** **27**

I **Appendix** **29**

A **Variable Length Coding** **30**

B **Error Reporting** **32**

C **Reserved Words** **32**

D **Core Language** **33**

E **Numerical Limits** **33**

F **Numerical Constants** **33**

Glossary **35**

Acronyms **35**

1 Motivation

This paper presents an approach to serializing objects, which is tailored for usability, performance and portability. In order to achieve these goals, in contrast to XML, we will sacrifice generality and human readability of the serialized format. Unlike other general serialization mechanisms, we provide explicit support for extension points in the serialized data, in order to provide a maximum of upward compatibility and extensibility.

The primary target of the proposed serialization mechanism are users which have to provide several tools with large amounts of data in a type safe and fast manner, but without the To do (5)

To do (6)

1.1 Related Work

There are many approaches similar to ours, but most of them have a different focus. This section shall provide a concise list of related approaches. For potential users of skill, this might also present alternatives superior for individual use cases.

XML

XML is a file format and might in fact be used as a backend. If a human readable storage on disk is not required, a binary encoding can be used to improve load/store performance significantly.

In contrast to our approach, XML requires the serialized data to form a tree. In theory, this is not a real problem, because there is at least one canonical transformation, which turns a graph into a tree (by adding a node and some attributes). On the other hand, such a transformation will in most cases take away the readability of the serialized data, which makes XML pointless.

XML Schema definitions

The description language itself is more or less equivalent to most schema definition languages such as XML Schema . The downside is that schema definitions have to operate on XML and can not directly be used with a binary format. There is also no way to generate code for some client languages, including Ada, from a schema definition. To do (7)

JAXP and xmlbeansxx

For Java and C++, there are code generators, which can turn a XML schema file into code, which is able to deal with an XML in a similar way, as it is proposed by this work. In case of Java this is even in the standard library. The downside is, that, to our knowledge, this is only possible for Java and C++, thus it leaves us with portability issues. A minor problem of this approach is the lack of support for comment generation and the inefficient storage of serialized data. An interesting observation is, that this approach deprives XML of its flexibility advantage over our solution. To do (8)

ASN.1

Is not powerful enough to fit our purpose.

IDL

A concise description of IDL can be fonud in . It seems not to be powerful enough and is certainly outdated. It is so old, that there are no bindings for any modern language. There is also not much documentation on further research on that area, thus creating a new approach with similar goals but modern techniques is in fact an option. To do (9)

Apatche Thrift & Protobuf

Lacks subtypeing. Protobuf has a overly complex notation language. Both seem to be optimized for network protocols, thus they do not have storage pools, which are the foundation of our serialization approach and an absolute requirement for some of our features, such as hints (see section 5.2).

Language Specific

Language specific is language specific and can therefore not be used to interface between subsystems written different programming languages such as Ada, Java, C or Haskell. Plus not every language offers such a mechanism. E.g. C.

Language Interfaces

Language Interfaces do not permit serialization capabilities. Most language only provide interfaces for C, with varying quality and varying degree of automation. A significant problem are interfaces between languages with different memory models. Interfaces between languages with different type systems are simply unproductive:D

2 Syntax

We use the tokens <id>, <string>, <int> and <comment>. They equal C-style identifiers, strings, integer literals and comments respectively. We use a comment token, because we want to emit the comments in the generated code, in order to integrate nicely into the target languages documentation system.

2.1 The Grammar

The grammar of a Serialization Killer Language (SKilL) definition file is defined as:

```
UNIT :=
  INCLUDE*
  DECLARATION*

INCLUDE :=
  ("include"|"with") <string> ";"?

DECLARATION :=
  DESCRIPTION
  <id>
  ((":"|"with"|"extends") <id>)?
  "{" FIELD* "}"

FIELD :=
  DESCRIPTION
  (CONSTANT|DATA) ";"?

DESCRIPTION :=
  (RESTRICTION|HINT)*
  <comment>?
  (RESTRICTION|HINT)*

RESTRICTION :=
  "@" <id> ("(" (R_ARG ("," R_ARG)*)? ")")? ";"?

R_ARG := ("%"|<int>|<string>)

HINT := "!" <id> ";"?

CONSTANT :=
  "const" TYPE <id> "=" <int>

DATA :=
  "auto"? TYPE <id>

TYPE :=
  ("map" MAPTYPE
  |"set" SETTYPE
  |"list" LISTTYPE
  |ARRAYTYPE)

MAPTYPE :=
  "<" GROUNDTYPE ("," GROUNDTYPE)+ ">"

SETTYPE :=
  "<" GROUNDTYPE ">"

LISTTYPE :=
  "<" GROUNDTYPE ">"

ARRAYTYPE :=
  GROUNDTYPE
  ("[" (<id>|<int>)? "]")?

GROUNDTYPE :=
  (<id>|"annotation")
```

Note: The Grammar is LL(1).[1]

Comment: The optional ; at the end of includes or definitions are for convenience only.

2.2 Reserved Words

The language itself has only the reserved words **annotation**, **auto**, **const**, **with**, **map**, **list** and **set**.

To do (10)

However, it is strongly advised against using any identifiers which form reserved words in a potential target language, such as Ada, C++, C#, Java, JavaScript or Python.

To do (11)

2.3 Examples

Listing 1: Running Example

```
/** A source code location. */
SLoc {
  i16 line;
  i16 column;
  string path;
}

Block {
  SLoc begin;
  SLoc end;
  string image;
}
```

[1] In fact it can be expressed as a single regular expression.

```
IfBlock : Block {
  Block thenBlock;
}

ITEBlock : IfBlock {
  Block elseBlock;
}
```

Includes, self references

Listing 2: Example 2a

```
with ”example2b.skill”

A {
  A a;
  B b;
}
```

Listing 3: Example 2b

```
with ”example2a.skill”

B {
  A a;
}
```

Unicode

The usage of non ASCII characters is completely legal, but discouraged.

Listing 4: Unicode Support

```
/** some arguably legal unicode characters. */
ö {
  ö ∀;
  ö €;
}
```

3 Semantics

This section will describe the meaning of individual keywords.

3.1 Includes

The file referenced by the with statement is processed as well. The declarations of all files reachable over `with` statements are collected, before any declaration is evaluated.

3.2 `annotation`

The type has a tag and a size, which allows it to be inserted at any annotation locations. This is useful in order to provide extension points in the file format. The file will still be readable by older implementations, which are not able to map any meaningful type into the annotation. A language binding is expected to provide something like an annotation proxy, which is used to represent annotation objects. If an application tries to get the object behind the proxy for an object of an unknown type, this will inevitably result in an error or exception. Therefore language bindings shall provide means of inspecting whether or not the type of the object behind an annotation is known.

As we will see in section 6, annotations are roughly equivalent to the type definition

```
annotation {
  v64 baseTypeName;
  v64 basePoolIndex;
}
```

Of course, this is made transparent to the user and some language bindings will offer a special and type safe treatment of annotations.

An implementation may treat an annotation pointing to an object of unknown type like a null reference. This behavior is safe, because such an object can not exist in the serialized file, thus the annotation has not been updated upon removal of the complete type pool. This behavior might look rather strange at first glance but is an effect of lazy treatment of informations stored in skill files and completely safe.

3.3 Sub Types

A sub type of a user type can be declared by appending the keyword `with` and the super types name to a declaration. In order to be well-formed, the sub type relation must remain acyclic and must not contain unknown types.

3.4 `const`

A const field can be used in order to create guards or version numbers, as well as overwriting deprecated fields with e.g. zeroes. The deserialization mechanism has to report an error if a constant field has an unexpected value.

3.5 `auto`

The language binding will create a field with the given type, but the content is transparent to the serialization mechanism. This is useful if the inference of the content of a field is likely to be faster then storing it, e.g. if it can be inferred lazily.

3.6 Abstract Data Types

To do (12)

Abstract Data Types (ADTs) showed to be useful and to increase the usability and understandability of the resulting code and file format.

ADTs are represented using arrays and pairs. To do (13)

The type system has a built-in notion of arrays, maps, lists and sets. Note that all of them are, from the view of serialization, equivalent to length encoded arrays. Their purpose is to increase the usability of the generated Application Programming Interface (API).

3.7 Comments

Comments provided in the skill file will be emitted into the generated code[2], thus allowing a user to get tool-tips in his IDE showing him this documentation. To do (14)

4 The Type System

To do (15)

User types can be seen as nonempty tuples over all types. Built-in types can be wrapped in order to give them special semantics. E.g. a time stamp can be created by:

Listing 5: Time

```
time {
  /** seconds since 1.1.1970 0:00 UTC. */
  i64 date;
}
```

Common Abbreviations

We will use some common abbreviations for sets of types in the rest of the manual:

Let ...

... $\mathcal{T}$ be the set of all types.

... $\mathcal{U}$ be the set of all user types.

... $\mathcal{I}$ be the set of all integer types, i.e. $\{\texttt{i8}, \texttt{i16}, \texttt{i32}, \texttt{i64}, \texttt{v64}\}$.

... $\mathcal{B}$ be the set of all built-in types.

[2] If the target language does not allow for C-Style comments, the comments will be transformed in an appropriate way.

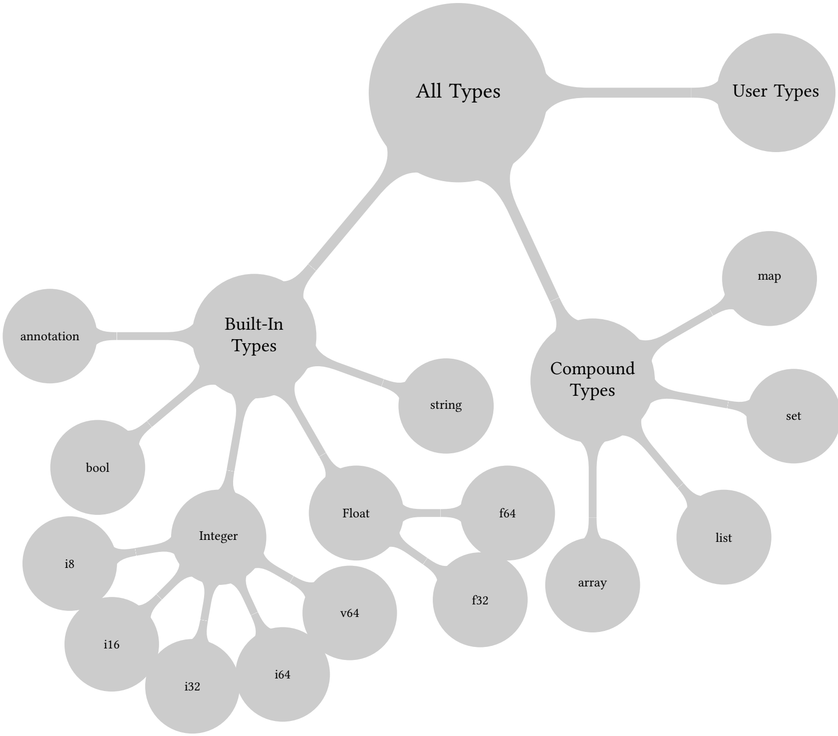

Figure 1: Layout of the Type System

4.1 Legal Types

The given grammar of SKilL already ensures that intuitive usage of the language will result in legal type declarations. The remaining aspects of illegal type declarations boil down to ill-formed usage of type and field names and can be summarized as:

- Field names inside a type declaration must be unique inside the type and all its super types[3].
- The subtype relation is a partial order[4] and does not contain unknown types.
- For all fields f of dependent array type[5], the size of the array has to denote a field of integer type in the very same declaration. The order of declaration is irrelevant.
- Any base type has to be known, i.e. it is either a ground type or it is a user type defined in any document transitively reachable over include commands.

4.2 Type Order

Let $<_l$ be the lexical order. We define a partial order $\leq_t$ on $\mathcal{T}$ as follows:

- $\forall t \in \mathcal{B}, s \in \mathcal{T} \setminus \mathcal{B}.t \leq_t s$

[3] The super type restriction may in fact be dropped?

[4] In fact it forms a forest.

[5] E.g. a field `t[size] f` requires another field of integer type in the same declaration – e.g. `i8 size`

- $\forall t \in \mathcal{C}, s \in \mathcal{U}. t \leq_t s$
- $\forall s, t \in \mathcal{U}. t \leq_t s \leftarrow s <: t$[6]
- $\forall s, t \in \mathcal{U}. t \leq_t s = t \leq_l s \leftarrow \exists S \in \mathcal{U} \cup \{\bot\}. t <: S \wedge s <: S$[7]

The informal short description is, first ground types, then compound types and user types at the end, where the forest of user types maintains its structure but is order using the lexical order of type names.

Notice, that this order corresponds to an left to right order in the types overview picture.

The missing order of compound types is left away intentionally, because it allows for the exchange of some type definition after publishing a format, e.g. `t[] f` can be exchanged with `list<t> f`.

4.3 Strings

Strings are conceptually a variable length sequence of utf8-encoded unicode characters. The in memory representation will try to make use of language features such as java.lang.String or std::u16string. The serialization is described in section **??**. If a language demands 0-termination in strings, the language binding will ensure this.

Strings should not contain 0 characters, because this may cause problems with languages such as C.

4.4 Compound Types

The language offers several compound types. Sets, Lists and auto sized Arrays, i.e. arrays without an explicit size, are basically views onto the same kind of serialized data, i.e. they are a length encoded list of elements of the supplied base type. Arrays are expected to have a constant size, i.e. they are not guaranteed to be resizable. Sets are not allowed to contain the same element twice. All ADTs will be mapped to their closest representation in the target language, while preserving these properties. Maps are viewed as a representation of serializable partial functions. Therefore they can contain other map types as their second type argument, which is basically an instance of currying.

4.5 NULL Pointer

The null pointer is serialized using the index 0. Conceptually, null pointers of different types are different. In fact if an annotation is a null pointer, it still has a type. However, this detail should not be observable in most languages.

4.6 Examples

This section will present some examples of ill-formed type declarations and brief explanations.

Listing 6: Legal Super Types

```
EncodedString : string {
```

[6] This is *super types first*.

[7] Types with the same or no supertype are order lexically.

```
    string encoding;
}
```

Error: The built-in type “string” can not be sub classed.

5 Type Annotations

5.1 Restrictions

Some invariants can be added to declarations and fields. These invariants can occur at the same place as comments, but can occur in any number. Invariants start with an @ followed by a predicate. Each predicate has to supply a default argument %, such that using only default arguments would not imply a restriction. If multiple predicates are annotated, the conjunction of them forms the invariant. The set of legal predicates is explained below.

If predicates, which are not directly applicable for compound types are used on compound types, they expand to the contents of the compound types, if applicable. Otherwise the usage of the predicate is illegal. To do (16)

AsField

maps:

In verbindung mit singletons kann man map-felder API-seitig zu feldern der interfaces machen!:)

Dadurch können sich einzelne tools felder ein und ausblenden, die dann zu maps serialisiert werden.

EP fordert, dass es einen Mechanismus gibt, der es einem erlaubt im nachhinein felder einzubauen;

Dieser Mechanismus erlaubt es einem zu jedem Zeitpunkt felder einzublenden; die gefahr ist hier, dass die Declarationen sehr unübersichtlich werden!

Range

Range restrictions are used to restrict integers and floats. To do (17)

Applies to fields: Integer, Float.

Signature: `range(min, max)`: $\alpha \times \alpha \rightarrow \textit{bool}$

Defaults: obvious.

Listing 7: Examples

```
natural {
  @range(0,%)
  v64 data;
}
positive {
  @range(1,%)
  v64 data;
}
nonNegativeDouble {
  @range(0,%)
  f64 data;
}
```

NonNull

Declares that an indexed field may not be null.

Applies to Field: Any indexed Type.

Signature: `nonnull()`

Defaults: none.

Listing 8: Examples

```
Node {
  @nonnull Node[] edges;
}
```

Unique

Objects stored in a storage pool have to be distinct in their serialized form, i.e. for each pair of objects, there has to be at least one field, with a different value.

NOTE: This can cause difficulties in combination with sub-classing, because the uniqueness property must hold even on the part restricted to the topmost class declared to be unique.

Applies to Declarations of indexed types.

Signature: `unique()`

Defaults: none.

Listing 9: Examples

```
@unique Operator {
  string name;
}
@unique Term {
  Operator operator;
  Term[] arguments;
}
```

Singleton

There is at most one instance of the declaration.

Applies to Declarations.

Signature: `singleton()`

Defaults: none.

Listing 10: Examples

```
@singleton System { ... }
@singleton Data{
  /** Note: if data would not be a singleton itself, it
      is likely to violate the singleton property */
  System foo;
}
```

Ascription

A language specific type can be ascribed to a field. The type has to be compatible to the fields actual type, because the ascription will not change the ABI in any way. The first argument is the language name. The second type is generator dependent, but should be related to types as they occur in local variable or field declaration in the respective language.

Although this kind of restriction puts a heavy burden on the language generator and decreases readability a lot, it can be used to increase the usability of the generated interface a lot, because language features such es enums in Java or unions and bitfields in C++ can be used.

Applies to fields.

Signature: `as(language, type)`: `string` × `string`→{}

Defaults: not allowed.

Listing 11: Examples

```
System {
  /**
  The language binding makes use of an enumeration,
     which is supplied with the generated code.

  The C++ interface will use the different type using
     C—Casts to convert between the two types (which is
     completely fine if the enum uses char as a base
     type).

  The Java interface will assume the stored integer to
     be the ordinal of the enum SystemState.
  */
  @as(”C++”, ”ccast␣SystemState”)
  @as(”Java”, ”enum␣SystemState”)
  i8 state
}
```

Tree

The reference graph below created by objects of this type forms a tree. The type of the objects is irrelevant. Strings and fields with notree annotation, are not taken into account.

Applies to Declarations or Field.

Signature: `tree()`

Defaults: none.

notree

Applies to field.

Signature: `notree()`

Defaults: none.

Listing 12: Examples

```
Sloc{...}
@tree
SyntaktikEntity{
  /** not a tree, because several entities, might share
      them */
  @notree Sloc sloc;

  SyntaktikEntity[] children;
}
Routine {
  @notree
  Routine[] callers;
  @tree
  Routine[] dominators;
}

@tree
File {
  File[] children;
  /** several files could have the same name,
        but strings are implicitly @notree */
  string name;
  string content;
}
```

Note: In case of the File example, there is no way to violate the tree property. Note: It is legal for trees to form forests.

5.2 Hints

Hints are annotations that start with a single ! and are followed by a hint name. Hints are used to control the behavior of the generated language binding and do not have an impact on the semantics of the stored data. Therefore they will not be stored in the reflection pool.

Access

Try to use a data structure that provides fast (random) access. E.g. an array list.

Modification

Try to use a data structure that provides fast (random) modification. E.g. a linked list.

Unique

Serialization shall unify objects with exactly the same serialized form. In combination with the @unique restriction, there shall at most be an error reported on deserialization.

Distributed

Use a static map instead of fields to represent fields of definitions. This is usually an optimization if a definition has a lot of fields, but most use cases require only a small subset of them. Because hints do not modify the binary compatibility, some clients are likely to define the fields to be distributed or even lazy. Note that this will increase both the memory footprint and the access time for the given field and will only be a benefit for memory-cache locality reasons. The internal representation will change from `f.a`, i.e. a regular field, to `pool.a[f]`, i.e a map in the storage pool which holds the field data for each instance. Note that the presence of distributed, lazy or ignored fields will require objects to carry a pointer to their storage pool, which may eliminate the cache savings completely.

Lazy

Deserialize the fields data only if it is actually used. Lazy implies distributed.

ReadOnly

The generated code is unable to modify the respective field or instances of the respective type. This options is provided to provide a consistent API while preventing from logical errors, such as modifying data from a previous stage of computation.

Ignore

The generated code is unable to access the respective field or any field of the type of the target declaration. This will lead to errors, if it is tried nonetheless. This option is provided to provide a consistent API for a combined file format, but restrict usage of certain fields, which should be transparent to the current stage of computation.

6 Serialization

This section is about representing objects as a sequence of bytes. We will call this sequence *stream*, its formal Type will be named S, the current stream will be named s. We will assume that there is an implicit conversion between fixed sized integers[8] and streams. We also make use of a stream concatenation operator $\circ : S \times S \longrightarrow S$.

This section assumes, that all objects about to be serialized are already known. It further assumes, that their types and thus the values of the functions (i.e. baseTypeName, typeName, index, $\llbracket\_\rrbracket$) explained below can be easily computed.

The serialization function $\llbracket\_\rrbracket_{\tau} : \tau \times \mathcal{T} \longrightarrow S$ will be written simply as $\llbracket\_\rrbracket$ if τ is clear form the context.

6.1 Steps of the Serialization Process

In general it is assumed that the serialization process is split into the following steps:

1. All objects to be serialized are collected. This is usually done using the transitive closure of an initial set.
2. The items are organized into their storage pools, i.e. the index function is calculated. To do (18)
3. The output stream is created as described below.

6.2 General File Layout

The file layout is optimized for lazy loading of stored data. It does also support type-safe and consistent treatment of unknown data structures. In order to achieve this, we have to store the type system used by the file together with the stored data. The type system itself is using strings for its representation, thus we have motivated the following layout.

```
utf8[][] stringPool

while(!EOF){
  string typeName
  string superTypeName
  option(v64 basePoolStartIndex; iff has superType)
  v64 sizeCount
  [[restrictions]]
  v64 fieldCount
  foreach f in fields {
    [[f.restrictions]]
    [[f.type]]
    string f.name
    @as(f.T[sizeCount])
    i8[] f.elements
  }
}
```

[8] As well as between fixed sized floating point numbers, because we define them to be IEEE-754 encoded 32-/64-bit sequences.

6.3 Storage Pools

This section contains the serialization function for an individual storage pool. We assume that storage pools are not empty. If an empty storage pool would be written to disk, it is simply skipped.[9]

Writing objects of a pool requires the following functions: $baseTypeName : \mathcal{U} \longrightarrow S$, $typeName : \mathcal{U} \longrightarrow S$ and $index : \mathcal{U} \cup \{\textbf{string}\} \longrightarrow S$.

The basic idea behind the serialization format is to store the data grouped by type into storage pools. If objects are referred to from other objects, those references are given as an integer, which is interpreted as index into the respective storage pool. The NULL pointer is represented by the index 0.

Each pool keeps a start index, which allows for the reconstruction of the complete object. A short example shall illustrate the basic concept. It contains five types A,B,C,D and N. Each has a single field of type τ which is used to simplify the representation. The type information for the objects in the base type pool can be inferred from the data stored in the pools using the links between the base type pool and the subtype pools (The base type start index (BPSI) field of pools with a super type – shown as arrows). For the sake of readability, the name, size and count fields are omitted in the picture.

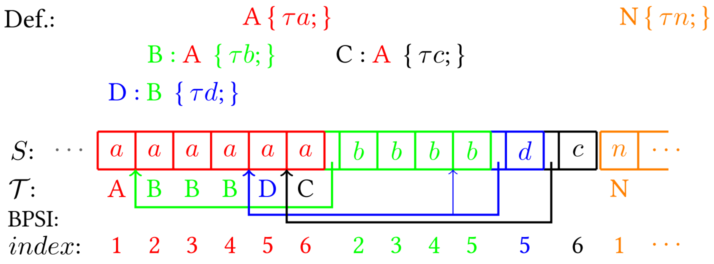

Figure 2: The serialization scheme used to store objects into pools.

The order in which pools are serialized is currently unrestricted.

6.4 Pool Elements

In this section, we want to describe the serialization of individual fields using the function $[\![\_]\!]_{\mathcal{T}}$. The serialization of an objects takes places by serializing all its fields into the stream. In this section, we assume that the three functions defined in the last section are implicitly converted to streams using the v64 encoding. We assume further, that compound types provide a function $size : \mathcal{T} \longrightarrow \mathcal{I}$, which returns the number of elements stored in a given field. Let f be a field of type t, then $[\![f]\!]$ is defined as[10]

- $\forall t \in \mathcal{U} \cup \{\textbf{string}\}.[\![f]\!]_t = \begin{cases} \texttt{0x00}, & f = \texttt{NULL} \\ index(f) & else \end{cases}$

- $[\![f]\!]_{\textbf{annotation}} = \begin{cases} \texttt{0x00 0x00}, & f = \texttt{NULL} \\ baseTypeName(f) \circ index(t) & else \end{cases}$ [11]

[9] This has the side effect, that only type information of instantiated types are present.

[10] We will use C-Style hexadecimal integer literals for integers in streams.

[11] We do not want to use type IDs here, because we do not want to touch all annotation fields if we modify the type Pool.

- $[\![\top]\!]_{\textbf{bool}} = \texttt{0xFF}$
- $[\![\bot]\!]_{\textbf{bool}} = \texttt{0x00}$
- $\forall t \in \mathcal{I} \setminus \{\textbf{v64}\}.[\![f]\!]_t = f$
- $[\![f]\!]_{\textbf{v64}} = encode(f)$[12]
- $[\![f]\!]_{\textbf{f32}} = [\![f]\!]_{\textbf{f64}} = f$[13]
- $\forall g \in \mathcal{B}, n \in \mathbb{N}^+.t = g\texttt{[}n\texttt{]} \implies [\![f]\!] = [\![f_0]\!]_g \circ \cdots \circ [\![f_{n-1}]\!]_g$
- $\forall g \in \mathcal{B}, s \in \mathcal{I}, \texttt{s size}$[14] $.t = g\texttt{[size]} \wedge \texttt{size} > 0 \implies [\![f]\!] = [\![f_0]\!]_g \circ \cdots \circ [\![f_{\texttt{size}-1}]\!]_g$[15]
- $\forall g \in \mathcal{B}, n = size(f), t \in \{g\texttt{[]}, \texttt{set<}g\texttt{>}, \texttt{list<}g\texttt{>}\}.[\![f]\!] = [\![n]\!]_{v64} \circ [\![f_0]\!]_g \circ \cdots \circ [\![f_{n-1}]\!]_g$
- Maps are serialized from left to right by serializing the keyset and amending each key with the map structure which it points to. In case of Maps with two types, this is equal to a list of key value tuples. A field of type `map<T,U,V>` is serialized using a schema $[\![size(f)]\!] \circ [\![f.t_1]\!] \circ [\![size(f[t_1])]\!] \circ [\![f[t_1].u_1]\!] \circ [\![f[t_1][u_1]]\!] \circ [\![f[t_1].u_2]\!] \circ \cdots \circ [\![size(f[t_2])]\!] \circ \cdots \circ [\![f[t_n][u_m]]\!]$. Note that we treat maps like map<T, map<U,V».
- $[\![RESTRICTION]\!] = \begin{cases} \emptyset, & id = \bot \\ [\![id]\!]_{v64}[\![arg_1]\!]_{string} \circ \cdots \circ [\![arg_n]\!]_{string} & else \end{cases}$
- $[\![t]\!]_{type} = \begin{cases} [\![id]\!]_{i8} \circ [\![val]\!]_t & id \in [0,4] \\ [\![id]\!]_{i8} & id \in [5,14] \\ 15 \circ [\![i]\!]_{v64} \circ [\![T]\!] & t = T[i] \\ 16 \circ [\![f.nameIndex]\!]_{v64} \circ [\![T]\!] & t = T[f] \\ 17 \circ [\![T]\!] & t = T[] \\ 18 \circ [\![T]\!] & t = list < T > \\ 19 \circ [\![T]\!] & t = set < T > \\ 20 \circ [\![n]\!]_{v64} \circ [\![T_1]\!] \circ \cdots \circ [\![T_n]\!] & t = map < T_i, \ldots, T_n > \\ [\![21 + reflectionPoolIndex(t)]\!]_{v64} & t \in \mathcal{U} \end{cases}$

6.5 Endianness

Files are stored in a little endian format, which is the default for common architectures.

If a client is running on a big endian machine, the endianness has to be corrected, both when reading and writing files. This can be done by changing the implementation of $[\![\_]\!]_{i*}$- and $[\![\_]\!]_{f*}$-translations.

[12] With encode as defined in listing 17.

[13] Assuming the float to be IEEE-754 encoded, which allows for an implicit bit-wise conversion to fixed sized integer.

[14] As stated above, size must be a field of the same declaration as f.

[15] Note that this is the only case where the encoded field does not append anything to the stream.

7 Deserialization

Deserialization is mostly straight forward.

The general strategy is:

- the string pool is deserialized into an array
- the reflection pool is deserialized using the strings array
- the structure of storage pools is read, pools are created and chunks of field data are copied into memory
- required fields are parsed using the information from the reflection pool

Date Example

Let d be the deserialization function – basically the inverse function of $[\![\_]\!]$.

$d(010464617465010002000100 0B010A01FFFFFFFFFFFFFFFFFF)$
$= d(01)d(04646174650100020001000B010A01FFFFFFFFFFFFFFFFFF)$
$= d(01)d(04)d(646174650100020001000B010A01FFFFFFFFFFFFFFFFFF)$
$= d(01)d(04)d(64617465)d(0100020001000B010A01FFFFFFFFFFFFFFFFFF)$
$= string[1 : ”date”]d(0100020001000B010A01FFFFFFFFFFFFFFFFFF)$
$= string[1 : ”date”]d(01)d(00)d(02)d(00)d(01)d(000B010A01FFFFFFFFFFFFFFFFFF)$
$= string[1 : ”date”]date[T : date[\_]1 : \_2 : \_d(000B010A01FFFFFFFFFFFFFFFFFF)$
$= string[1 : ”date”]date[T : date[d(000B01)]1 : \_2 : \_d(0A01FFFFFFFFFFFFFFFFFF)$
$= string[1 : ”date”]date[T : date[v64date]1 : \_2 : \_d(0A01FFFFFFFFFFFFFFFFFF)$
$= string[1 : ”date”]date[T : date[v64date]1 : \_2 : \_d(01FFFFFFFFFFFFFFFFFF)]$
$= string[1 : ”date”]date[T : date[v64date]1 : [1]2 : [-1]]$

8 In Memory Representation

This section is to describe the API provided to the programmer and the representation of objects inside memory. Both are not mandatory and can depend on the target language and implementation.

8.1 API

The generated API has to be designed in a fashion that integrates nicely with the languages programming paradigms. E.g. in Java it would be most useful to create a state object, which holds state of a bunch of serializable data and provides iterators over existing objects, as well as factory methods and methods to remove objects form the state object. The serialized types can be represented by interfaces providing getters, setters, using hidden implementations, only known to the state object.

talk about the generated API and its features, like iterators, factories, access to singletons and stuff.

Examples

Nice example in C++:

Listing 13: C++ Examples

```
#include <stdint.h>
#include <string>
[...some other bouilerplate includes...]
struct SLoc {
  uint16_t line;
  uint16_t column;
  std::string* path;
};
struct Block {
  std::string* tag;
  SLoc* begin;
  SLoc* end;
  std::string* image;
};
struct IfBlock : public Block {
  Block thenBlock;
};
struct ITEBlock : public IfBlock {
  Block elseBlock;
};
[...
  plus some boilerplate code for visitors, iostreams etc.
...]
```

Listing 14: Java Examples

```
class SLoc {
  public short line;
  public short column;
  public String path;
}
class Block {
  final public String tag() {
    return this.getClass().getName();
  }
  public SLoc begin;
  public SLoc end;
  public String image:
}
class IfBlock extends Block {
  public Block thenBlock;
}
class ITEBlock extends IfBlock {
  public Block elseBlock;
}
[...some read and write code, plus some visitors...]
```

Listing 15: LaTeX Examples

```
$(line, column, path) \in SLoc
  \subseteq \mathbb{Z} \times \mathbb{Z} \times string$

$(begin, end, image) \in Block
  \subseteq SLoc \times SLoc \times string$

$(super, thenBlock) \in IfBlock
  \subseteq Block \times Block$

$(super, elseBlock) \in ITEBlock
  \subseteq IfBlock \times Block$
```

Which looks like:

$(line, column, path) \in SLoc \subseteq \mathbb{Z} \times \mathbb{Z} \times string$
$(begin, end, image) \in Block \subseteq SLoc \times SLoc \times string$
$(super, thenBlock) \in IfBlock \subseteq Block \times Block$
$(super, elseBlock) \in ITEBlock \subseteq IfBlock \times Block$

Note: The incentive of the LaTeX-output is to provide a mechanism for users to formalize their file format using mechanisms, that are or can not be available as a specification language. E.g. the sentence “The path of a SLoc points to a valid file on the file system and the line and column form a valid location inside that file.” can not be verified in a static manner. This is because the correctness of the property depends not only on the content to be verified, but on the verifying environment as well.

8.2 Representation of Objects

The combination of laziness and consistency has the effect, that representation of objects inside memory is rather difficult. This section describes data structures and algorithms which basically do the job. In this section, we assume that all fields are present as arrays of bytes. We will describe the effects of parsing fields in an unmodified state, in a modified state, how to modify a state and finally how to write a state back to disk.

8.2.1 Proposed Data Structures

A state has to contain at least these informations:

- an array of strings
- the type information
- storage pools

A storage pool has to hold the images of fields, which are not yet parsed.

Objects are required to have an ID field, which corresponds to the ID of the deserialized(!) state. This field is required in order to map the lazy fields to the correct objects. It can also be reused in the serialization phase to assign unique IDs, which will be used instead of pointers.

Objects of types with eager fields should have the respective fields. E.g. the declaration `T {t a; !lazy t b;}` should be represented by an object

```
InternalTObject {
  long ID;
  t a;
  /* getA, setA... */

  StoragePool tPool;
  /* getB, setB... */
}
```

Note that the pointer to the enclosing storage pool is required for the correct treatment of lazy and distributed fields. This is because the pool holds the field data.

Now that we have a representation for objects, we still have to store objects across storage pools. The possibility of inheritance requires us to store objects in multiple pools at the same time. We propose to store super and sub type information inside pools as links to the pools of the respective types. The objects should be stored as a “double linked array list”, which is basically a linked list containing array lists:

Each pool stores the objects, with the static type of the pool in an array list. The pools of subtypes are stored in a linked list. Now an iterator over all elements of a type uses an iterator over all elements of a pool in combination with iterators over the pool and all its sub pools. Therefore creating and deleting objects is an amortized O(1) operation and it is guaranteed to maintain the semantic structure of the file, if IDs are not updated in phases other than reading and writing a file.

Note that, in the presence of distributed fields, the runtime complexity of these operations will change. It is expected to reduce to O(log(n)) where n is the largest number of objects with a common distributed field.

8.2.2 Reading Unmodified Data

Reading unmodified data is basically done by creating objects with ascending IDs and adding all eagerly processed fields to them. Pointer resolution in an unmodified state is an O(t) operation, where t is the size of the type hierarchy below the static type of the pointer.

During the reconstruction of the initial dataset, an array in the base pool may be used to reduce the cost to O(1)[16]. However, this helper array has to be dropped, as soon as the base pool is modified.

8.2.3 Modifying Data

The only legal way of modifying data is to access it through the generated API, which provides iterators, a type safe facade, factories and means of removing objects from states for each known type. A modification is any operation that will invalidate any existing object ID, i.e. deleting objects or inserting objects into nonempty storage pools. Adding objects to empty storage pools does not count as a modification in the sense of a modified state, because it is not possible, that a pointer to such an object lures in an yet untreated field.

8.2.4 Reading Data in an Modified State

Reading Data in a modified state, is very similar to reading data in the unmodified state. Except that resolution of stored pointers can no longer rely on the trick that the ID of an object is also the index into the base type pool. There is a solution for this problem using O(t + log(n)), where n is the number of instances with the same static type. However, the straightforward implementation is O(t+n).

With this difference in mind, we strongly recommend adding a dirty flag to each storage pool which traces modifications. This will effectively eliminate the additional cost, because transformation of stored state is expected to be monotonic growing in a way that each step adds instances of a type, which was not present before and does not change old data.

8.2.5 Writing Data

Data which is lazy and modified can not be wrote to disk. Therefore any data which can potentially refer to modified data has to be evaluated. After evaluating the respective data, serialization is straightforward. The evaluation of lazy data referring to potentially modified data can be done in $O(out)$, where out is the size of the output file. Writing the output is also in $O(out)$, thus it is not per se a problem.

8.2.6 Final Thoughts on Runtime Complexity

Although the last sections read a lot like accessing serializable data being unnecessary expensive, this is in fact not the case.

Reading data without modifying it is $O(in)$, even if only a part of the data is read. This is mainly caused by the requirement of being able to process unknown data correctly. The actual cost should be limited by the cost of sequentially reading the input file from disk.

Reading data modifying it and writing it back is $O(in+m+out)$, which is not surprising at all, because one has to pay for reading the initial file, writing the complete output file and the modifications.

[16] The creation time for the array can be paid for during the creation of the base pool.

Only the usage of a lot of lazy or distributed data is expensive. It is generally advised against using the lazy attribute if a field is read for sure during the lifetime of a serializable state.

9 Case Study: Skill Encoded XML

Although it is not very clever to use skill for encoding xml files, because one basically looses all benefits from both worlds, we will do so as demonstration for the compression yielded by the skill serialization scheme. Honestly most effects will be obtained from strings being stored in the string pool. Because most of the validation mechanisms directly built into xml are not required in skill and for the sake of simplicity, we will strip xml to its bare payload:

Listing 16: Skill Encoded XML

```
XML {
  string xmlDecl;
  Element element;
}
Element {
  string name;
  map<string, string> attributes;
  /** @note we will supply the empty string, if no
          content is present */
  string content;
  Element[] children;
}
```

To do (19)

To do (20)

To do (21)

10 Future Work

XML output mit XML Schema.

Das neue Serialisierungsschema erlaubt es einen Viewer zu bauen, der Definition+Datei anzeigen kann. (Die future work ist hier der viewer)

Integration der Definition in die Serialisierte Form, damit man die Daten generisch prüfen und anzeigen kann. Hier braucht man noch ein gutes encoding, weil man sonst zu viel platz verbraucht.

Abuse annotations for type-safe unions. The type system does not allow for unrestricted unions or intersection types. The former violate serialization invariants, the latter would either have no instances or be equal to an already existing (super) type.

State somewhere, that a major advantage over XML is, that one is not required to link against a overly general implementation, which is nice if one is only interested in a very specific format.

A notion of *first class strings* can be used to separate the string pool into one pool at the beginning of the file which contains all the strings, that have to be used in order to understand the contents of the file and a second part of the pool, which can be skipped. This should give significant performance improvements if files with lots of unused strings are processed.

True comments with `# ... \n`?

Alternativ kann man die reflection data aus dem ReflectionPool als Eingabesprache für einen Generator benutzen.

Fun fact: Garbage Collection for serializable objects comes for free, if objects are always held in storage pools.

To do...

□ 1 (p. 1): blablabla

□ 2 (p. 1): leider wird man nicht um einen glossar rumkommen. ABI, API, super type, base type, ...

□ 3 (p. 1): man muss klar definieren, was groundtypes und was basetypes sind und die begriffe dann auch konsistent benutzen

□ 4 (p. 1): string in *unique string* o.ä. umbenennen, um verwirrung bei C Programmierern vorzubeugen

□ 5 (p. 3): hier muss noch was wegen version resilience, small footprint, den daten, die man so üblicherweise hat (1000 objekte eines typs, pointer, etc) und den mehrfachen sichten über verschiedene spezifikationen rein. Außerdem sollte man erwähnen, dass man hier den reflection mechanismus for free bekommt

□ 6 (p. 3): hier vielleicht sagen, dass die erwartete zahl von instanzen im bereich [10^2;10^9] liegt

□ 7 (p. 3): cite w3c

□ 8 (p. 3): brr

□ 9 (p. 4): ref David Lamb

□ 10 (p. 6): check for updates

□ 11 (p. 6): Appendix with a list of all identifiers which form reserved words in one of the languages above, including our keywords

□ 12 (p. 9): rewrite section; it emerged from the fusion of two sections talking about ADTs

□ 13 (p. 9): ref encoding scheme which will be explained later; somewhat confusing

□ 14 (p. 9): sprache!

□ 15 (p. 9): ein paar einleitende worte

□ 16 (p. 13): hier muss man zwischen serialisierbaren und nicht serialisierbaren restrictions unterscheiden; serialisierbar sind alle restrictions, die auch auswirkungen auf die potentiell gespeicherten daten haben, wie range und nonnull

□ 17 (p. 13): Falls $0 \notin [min, max]$ wird der default min.

□ 18 (p. 18): das ist nur die halbe wahrheit, weil man hier auch noch updates(im sinne von dynamischer logik:)) für modifikationen machen muss

□ 19 (p. 27): compare size of some svg files

□ 20 (p. 27): compare speed of load/store of those svg files

□ 21 (p. 27): some final comments to say, that the comparison is of course not completely fair, and that it is advised against mixing xml and skill in most cases

Part I

Appendix

A Variable Length Coding

Size and Length information is stored as variable length coded 64 bit unsigned integers (aka C's `uint64_t`). The basic idea is to use up to 9 bytes, where any byte starts with a 1 iff there is a consecutive byte. This leaves a payload of 7 bit for the first 8 bytes and 8 bits of payload for the ninth byte. This is very similar to the famous utf8 encoding and is motivated, as it is the case with utf8, by the assumption, that smaller numbers are a lot more likely. It has the nice property, that there are virtually no numerical size limitations. The following small C++ functions will illustrate the algorithm:

Listing 17: Variable Length Encoding

```
uint8_t* encode(uint64_t v){
  // calculate effective size
  int size = 0;
  {
    auto q = v;
    while(q){
      q >>= 7;
      size++;
    }
  }
  if(!size){
    auto rval = new uint8_t[1];
    rval[0]=0;
    return rval;
  }else if(10==size)
    size = 9;

  // split
  auto rval = new uint8_t[size];
  int count=0;
  for(;count<8&&count<size -1;count++){
    rval[count] = v >> (7*count);
    rval[count] |= 0x80;
  }
  rval[count] = v >> (7*count);
  return rval;
}
```

Listing 18: Variable Length Decoding

```
uint64_t decode(uint8_t* p){
  int count = 0;
  uint64_t rval = 0;
  register uint64_t r;
  for(;count<8 && (*p)&0x80; count++, p++){
    r = p[0];
    rval |= (r&0x7f)<<(7*count);
  }
  r = p[0];
  rval |= (8==count?r:(r&0x7f))<<(7*count);
  return rval;
}
```

B Error Reporting

This section describes some errors regarding ill-formatted files, which must be detected and reported. The order is based on the expected order of checking for the described error. The described errors are expected to be the result of file corruption, format change or bugs in a language binding.

Deserialization

- If `EOF` is encountered unexpectedly, an error must be reported before producing any observable result.
- If an index into a pool is invalid[17], an error must be reported.
- If the deserialization of a storage pool does not consume exactly the `sizeBytes` in its header, an error must be reported. Note: This is a strong indicator for a format change.
- If the serialized type order of storage pools does not match the expected type order, an error must be reported.
- If the serialized type information contains cycles, an error must be reported, which contains at least all type names in the detected cycle and the base type, if one can be determined.
- If a storage pools contains elements which, based on their location in the base pool, should be subtypes of some kind, but have no respective sub type storage pool, an error must be reported with at least, the base type name, the most exact known type name and the adjacent base type names.
- All known constant fields have to be checked before producing any observable result. If some constant value differs from the expected value, an error must be reported, which contains at least the type, the field type and name, the basePoolIndex, the index inside the types pool, the expected value and the actual value.
- If a serialized value violates a restriction or the invariant of a type,[18] an error must be reported as soon as this fact can be observed. It is explicitly not required to check all serialized data for this property.

C Reserved Words

This section contains a table of words which must not be used as field names, because they are keywords in some languages. The usage of skill keywords will result in a direct error, whereas the usage of a word listed below will result in a warning, because the identifier will be escaped in the target language binding.

if	**then**	**else**	**begin**	**end**
struct	**class**	**public**	**protected**	**private**
⇒	...			

[17] because it is larger then the last string in the pool

[18] Including sets containing multiple similar elements.

D Core Language

The core language is a subset of the full language which must be supported by any generator, which is called skill core language generator. Features included in the core language are:

- Integer types `i8` to `i64` and `v64`
- `string`, `bool` and `annotation`
- Compound types
- User Types with sub-typing
- `const` and `auto` fields.
- Reflection.

Thus the remaining parts required for full skill support are:

- Floats
- Restrictions
- Hints
- Language dependent treatment of comments, e.g. integration into doxygen or javadoc.[19]
- Name mangling to allow for usage of language keywords or illegal characters (unicode) in specification files, without making a language binding impossible.

E Numerical Limits

In order to keep serialized data platform independent, one has to respect the numerical limits of the various target platforms. For instance, the Java Virtual Machine will not allow arrays with a size larger then 2^31 minus some elements. Therefore we establish the following rule:

(De-)serialization of a file with an array of more then 2^30 elements or a type with more then 2^30 instances may fail due to numerical limits of the target platform.

F Numerical Constants

This section will list the translation of type IDs(see ??) and restriction IDs (see ??). Restrictions with undefined IDs will not be serialized.

[19] This may even require a language extension providing tags inside comments which are translated into tags of the respective documentation framework.

Type Name	Value
const i8	0
const i16	1
const i32	2
const i64	3
const v64	4
annotation	5
bool	6
i8	7
i16	8
i32	9
i64	10
v64	11
f32	12
f64	13
string	14
T[i]	15
T[f]	16
T[]	17
list<T>	18
set<T>	19
map<T_1, ..., T_n>	20
T	21 + $index_T$

(a) Type IDs

Restriction Name	Value
range	0
nonnull	1
unique	2
singleton	3
...	...

(b) Restriction IDs

Glossary

base type The root of a type tree, i.e. the farthest type reach able over the super type relation.. 16, 26

built-in type Any predefined type, that is not a compound type, i.e. annotations, booleans, integers, floats and strings.. 7, 26

ground type Any type, that is not a compound type, i.e. the union of user defined types and built-in types. 26

sub type If a user type A extends a type B, A is called the sub type (of B).. 6, 26

super type If a user type A extends a type B, B is called the super type (of A).. 6, 26

unknown type We will call a type *unknown*, if there is no visible declaration of the type. Such types must not occur in a declaration file, but they can be encountered in the serialization or deserialization process.. 6, 26

user type Any type, that is defined by the user using a type declaration. 6, 7, 26

visible declaration We will call a type declaration *visible*, if it is defined in the local file, or in any file transitively reachable over include directives.. 26

Acronyms

ABI Application Binary Interface. 26

ADT Abstract Data Type. 6, 26

API Application Programming Interface. 6, 18, 26

SKilL Serialization Killer Language. 3, 8, 13, 15, 26