The SKilL Language

Timm Felden

October 30, 2014

Abstract

This paper presents an approach to serializing objects, which is tailored for usability, performance and portability. 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.

Acknowledgements

Main critics: Erhard Plödereder and Martin Wittiger.
Additional critics: Dominik Bruhn[1], Dennis Przytarski[2].

Contents

[1]Discussion on SKilL and related work
[2]Discussion on API and proposal of reordering type information; Pragmas

Panta rhei (everything flows)

Heraclitus of Ephesus

Part I
Specification Language

To do (1)
To do (2)
To do (3)
To do (4)
To do (5)
To do (6)

1 Introduction

This section provides some introductory words on Serialization Killer Language (SKilL) and an overview on how to read this document.

1.1 Motivation

Many industrial and scientific projects suffer from platform or language dependent representation of their core data structures. These problems often cause software engineers to stick with outdated tools or even programming languages, thus causing a lot of frustration. This does not only increase the burden of hiring new project members, but can ultimately cause a project to die unnecessarily.

The approach presented in this paper provides means of platform and language independent specification of serializable data structures and therefore a safe way to let old tools of a tool suite talk to the new ones, without even the need of recompiling the old ones. We set out to design a new and easy to use way of making core data structures of a tool platform language independent, because we believe that the best language a programmer can use to write a new tool, is the language that he likes the most. We also had the strict requirement to provide a solution that can describe an intermediate representation with stable parts that can be used for decades and unstable parts that may change on a daily basis.

In order to achieve this goal, we introduce two new concepts:

The first one is an easy to use specification language for data structures providing simple data types like integers and strings, container types like sets and maps, type safe pointers, extension points and single inheritance. The specification language is modular in order to make large specifications more readable.

The second one is a formalized mapping of specified types to a bitwise representation of stored objects. The mapping is very compact and therefore scalable, easy to understand and therefore easy to bind to a new language. It does encode the type system and can therefore provide a maximum of upward and downward compatibility, while maintaining type safety at the same time. It allows for a maximum of safety when it comes to manipulating data unknown to the generated interface, while maintaining high decoding and encoding speeds[3].

An improvement over the Extensible Markup Language (XML), our main competitor, is that the reflective usage of stored data is expected to be quite rare, because the SKilL binding generator is able to generate an interface that ensures type safety of modifications and provides a nice integration into the target language. This leads to a situation, where it is possible to use files containing data of arbitrary types. If the data stored in the file is not used by a client, he does not have to pay for it with execution time or memory. Furthermore a client does not have to know the whole intermediate representation of a tool suite, but only the parts he is going to use in order to achieve his goals. The expected file sizes range from a megabyte to several gigabytes, while

[3]The serialization and deserialization operations are linear in the size of the input/output file.

having virtually no relevant numerical limits in the file format[4]. Please note, that the SKilL file format is a lot more compact then equivalent XML files would be. It is expected, that files contain objects of hundreds of types with thousands of instances each. If a type in such a file would contain three pointers on average, the file size would still be around a mega byte, which is due to a very compact representation of stored data. This will also lead to high load and store performance, because the raw disk speed is expected to be the limiting factor.

1.2 Scientific Contributions

This section is a very concise representation of contributions that in part have already been mentioned above and in parts will be mentioned much later.

The suggested serialization format and serialization language offer all of the following features in a single product:

- a small footprint and therefore high decoding speeds
- a fully reflective type encoding
- type safe storage of references both to known and unknown types[5]
- a rich type system providing amongst others references, containers, single inheritance and extension points
- the specification language is modular[6] and easy to use
- no tool using a common intermediate representation has to know the complete specification. It is even possible to strip away or add individual fields of commonly used types – independent of temporal, physical or lexical order.
- the coding is platform and language independent
- the coding offers a maximum of downward **and** upward compatibility
- a programmer is communicating through a generated interface, which allows programmers, knowing nothing about the SKilL, to interact with it, and ensures type safety. It also allows programmers to write tools in the language they know best[7].
- stored data, that is never needed by a tool, will never be touched.
- new objects can be added to a file by appending data to the existing file.

Any of the arguments above have already been made in various contexts (e.g. [TDB$^+$06] §13.13, [LA13], [xml06], [Lam87]), but there is, to the best of our knowledge, no solution bringing all these demands together into a single product that does the job automatically.

[4]There are practical limits, such as Java having array lengths limited to 2^{31} or current file systems having a maximum file size limit that is roughly equivalent to the size of a file completely occupied by objects with a single field of a single byte. There will also be problems with raw I/O-Performance for very large files and an implementation of a binding generator, which can handle files not storable in the main memory is a tricky thing to do.

[5]I.e. regular references and annotations.

[6]I.e. it can be distributed over many files.

[7]This is a problem especially in the scientific community, where many researchers work on similar problems but with completely different tools.

1.3 Outline

The specification has been split into three basic parts:

Part I will describe the specification language of SKilL. It is written mostly from the perspective of a user of SKilL with little detail on technical foundations.

Part II will describe our goals for APIs provided by SKilL bindings to users in arbitrary programming languages. This can be used by users to get a general grasp, but the target readership are designers of language bindings and code generators.

Part III will describe the binary file format used by SKilL. The text is written with implementers in mind and may be of little interest to most users.

The remainder is filled with an outlook on future developments as well as rather technical lists and tables that mainly provide a concise overview and clarification.

1.4 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 (defined in [xml06]). The main differences are:

+ XML can be manipulated with a text editor[8].
+ It is easier to write a libXML for a new language than to write a SKilL back-end[9].

- XML is not an efficient encoding in terms of (disk-)space usage. This can be overcome by the Efficient XML Interchange Format (EXI) (see [SK11]).
- XML is not type safe. This can be overcome partially by the XML Schema Definition Language (XSD).
- XML does not provide references to other objects out of the box.
- XML stores basically a tree, whereas a SKilL file contains an arbitrary amount of graphs of objects.
- XML is usually accessed through a libXML, whereas SKilL provides an API for each file format, thus a SKilL user does not require any SKilL skills, i.e. no knowledge about SKilL types or the representation of serialized objects is required in general. To be honest, there are some language bindings, mainly for Java, which offer this benefit for XML as well.
- XML-files can not be appended with new data.

[8]Whereas SKilL files are binary and require a special editor, which will be provided by us eventually.
[9]This is only a relevant point if no bindings exist for the language you want to use.

XML Schema definitions

SKilLs description language itself is more or less equivalent to XML schema definition languages such as XSD (as described in [GSMT$^+$08, PGM$^+$08]). The most significant difference is caused by the fact that XML operates on trees and SKilL operates on arbitrary graphs.

The type systems offered by SKilL and XSD are quite different, thus it might be worth a look which one better fits ones needs.

JAXP and xmlbeansxx

For Java and C++, there are code generators that turn an XML schema file into code, which is able to deal with an XML in a similar way as proposed by this work. In case of Java the mechanism is even part of the standard library. The downside is that, to the best of 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.

XML based Approaches

There are various XML based formats. All XML based formats share the basic performance overhead of XML. A close competitor in terms of goals is GXL[WKR02]. Publications on GXL seem to stop in mid 2002. An examination of example GXL coded graphs imply that the format is wasting too much space to be scalable for larger graphs as common in the process of analysing medium size programs. The TGraphs library[ERW08] claims to use GXL for communication with other tool chains.

ASN.1

Is not powerful enough to fit our purpose.

IDL

The published format is stated to be ASCII ([Lam87] §2.4), which will cause similar efficiency problems as raw XML does, if large amounts of data are stored. Note that the changes in architecture of computer systems over the last two decades makes the solution less appealing then it was at the time of its creation. Tools like memory mapped files or IEEE-754 floats did simply not exist at that time.

Apache Thrift, Protocol Buffers and others

Thrift states that there is no sub-typing (see [Apa13][10]). Protocol Buffers (see [Goo13]) seem not to support sub-typing either. Both seem to be a pragmatic approach to generalization of efficient network protocols. The type system of Protocol Buffers is also a rather pragmatic solution offering types such as unsigned 32 bit integers, which can not be represented in an efficient and safe way by e.g. Java. Both do not have storage pools, which are the foundation of our serialization approach and an absolute requirement for some of our optimizations, such as hints (see section 5.2).

[10]In section ”structs”, first sentence: ”Thrift structs define a common object – they are essentially equivalent to classes in OOP languages, but without inheritance.”, as of 29.Aug.2013

Protocol Buffers provide a variable length integer type, namely `int64`, which seems to be binary compatible[11] with the variable length integer type used in SKilL (see section C).

Lately some people seem to realize that the in-memory representation can be moved over the network without modification, if it is the same at both ends. Solutions of that kind include CapNProto[Var14] and SBE[TM13].

Solutions mentioned in this section have an advantage in terms of performance but offer little to no change tolerance at all. Therefore, they are best used in an environment with a lot of communication and total control over protocol changes. If protocol changes are frequent, the change tolerance provided by the SKilL method may save a significant amount of development costs.

Java Bytecode, LLVM/IR and others

Although Java Bytecode (see [LYBB13]) and the LLVM Intermediate Representation (see [LA13]) are hand crafted formats, they served as a guiding example in many ways.

Language Specific Serialization

Language specific serialization is language specific and can therefore not be used to interface between subsystems written in different programming languages, without a lot of effort. Our aim is clearly a language independent and easy to use serialization format.

Interesting enough is the fact that in Effective Java 2[Blo08] items 74 and 75 warn you not to use serialization if you are not completely sure about it and to use a hand written serialization mechanism if possible. The reason for both points is the lack of change tolerance in Java's built-in serialization mechanism. Those reasons do not apply to SKilL based serialization in most settings.

SKilL Implementations of past Revisions

This revision is based on experience with previos versions of skill. Implementations can be found under . Descriptions of implementations can be found in . To do (7)

To do (8)

1.5 Notation

Code and references to code will be written in a `type writer font`. Full pieces of code are grouped into listings. If the piece is part of the SKilL test suite, the heading of the block is the respective file name.

Listing 1: "Example Listing"

```
Some Code {
  ... <- more stuff that is not important at this point
}
```

Semantics and types will be written in *italics*. Sets of types like the set of all types are written in calligraphy, e.g. $\mathcal{T}$.

[11] The Protocol Buffer implementation seems not to optimize away the ninth flag, thus it might use an additional byte for very large numbers.

2 Syntax

This section discusses the syntax of the description language in brevity. The semantics is discussed in section 3, the file format is discussed in section 8.

We use the tokens `<id>`, `<string>`, `<int>`, `<float>` and `<comment>`. They equal C11-style [12] identifiers, strings, integer literals, float literals and comments respectively. Identifiers, strings and comments are explicitly enriched by printable Unicode characters above `\u007f`, although this feature should be used with care. For the sake of portability Unicode characters are restricted to 16 bit code points only. Usage of code points above `\uffff` has no defined behaviour[13]. We use a comment token, because we want to emit the comments in the generated code, in order to integrate nicely into the target language's documentation system.

2.1 Reserved Words

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

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, as well as the identifiers **skillid**, **internal** and **api**.

2.2 The Grammar

A simplified overview of the grammar of a SKilL definition file is given below. For the sake of readability, fancy types, change modifiers and comments have been simplified or removed. The complete grammar can be found in appendix A. The detailed explanation of individual syntactic constructs will make use of the complete grammar. To do (9)

To do (10)

```
UNIT :=
  TrueComment*
  INCLUDE*
  DECLARATION*

TrueComment := ("#" ~[\n]* \n)*

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

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

FIELD :=
  DESCRIPTION
```

[12][ISO11] Annex D

[13] A code generator is expected to reject specifications containing these characters, whereas implementations are expected to treat user strings containing 32 bit code points correctly.

```
  (CONSTANT|DATA) ";"

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

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

R_ARG := (<float>|<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
  ("[" (<int>)? "]")?

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

Note: The Grammar is LL(1).[14]

To do (11)

To do (12)

2.3 Examples

Listing 2: Running Example

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

```
/** A location in a file pointing to a character in that
    file. Assumes ordinary text files. */
Location {

  /** the line of the character starting from 0 */
  i16 line;

  /** the column of the character starting from 0 */
  i16 column;

  /** the file containing the location */
  File path;
}

/** A range of characters in a file. */
Range {

  /** first character; inclusive */
  Location begin;

  /** last character; exclusive */
  Loctaion end;
}

/** A hierarchy of file/directory names. */
File {

  /** Name of this file/directory. */
  string name;

  /** NULL iff root directory. */
  File directory;
}
```

Includes, self references

Listing 3: Example 2a

```
with "example2b.skill"

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

Listing 4: Example 2b

```
with "example2a.skill"
```

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

which is equivalent to the file:

Listing 5: Example 2

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

B {
  A a;
}
```

Subtypes

Types can be extended using subtyping:

Listing 6: Subtyping Example

```
with "runningExample.skill"

/** a message is just a string */
Message {
  string message;
}

/** located messages contain a location as well */
LocatedMessage extends Message {
  Location location;
}
```

Containers

Container types can be used to store more elaborate data structures, then just plain values or references. In the current version, there is support for sets, maps, lists and arrays:

Listing 7: Containers

```
/** E.g. a user in a social network. */
User {
  string name;

  /** friends of this user */
  List<User> friends;
  /** default values of permissions can be overriden on
     a per-user basis. The value is stored explicitly to
     ensure that the override survives changes of the
     permissions default value. */
  Map<User, Permission, Bool> permissionOverrides;
}

Permission {
  string name;
  bool default;
}
```

Unicode

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

Listing 8: Unicode Support

```
Ä {
  Ä  ;
  Ä €;
}
```

2.4 Style Guide

This section provides some general hints on how to write readable specifications. A uniform appearance of specifications will improve the readability and therefore reduce development time.

Use speaking names Although type and field names are serialized, the overhead is a small constant. Furthermore, speaking names tend to prevent name clashes and decrease development time.

Use camelCase identifiers consisting of letters a-z only This rule enables code generators to create readable names for target languages. Letters a-z can be represented in most encodings, are used by the English language and can therefore assumed to be safe. Usage of other characters may require unreadable escape sequences.

If Acronyms have to be used followed by a word then the last capitalized character is interpreted as the first character of the subsequent word.

Use Capitalized user type names This convention is used by most programmers. If this convention contradicts the style guide or rules of a language, the language specific binding generator is expected to change casing according to the respective style guide.

Use lowercase field type names Field Types should consist of a single word only. This convention is used by most programmers. If this convention contradicts the style guide or rules of a language, the language specific binding generator is expected to change casing according to the respective style guide.

Indent Units with two spaces Specifications are likely to be processed with regular text editors. The lack of control flow allows for a small indention level.

Comments, Restrictions and Hints should reside in their own lines Using separate lines for type or field modifications causes the amount of modification to be reflected in the visual appearance of the entity.

Separate field declarations with comments, restrictions or hints with one blank line from other field declarations This ruled turned out to be very usable. If neither comments, hints or restrictions are used, the specification may choose to skip blank lines. Not using comments is discouraged. Separating subsequent type definitions with two blank lines turned out to be useful as well.

Declare super types first The skill specification language itself enforces no order on type declarations. Human readers however may not be able to read specifications where super types are used before they are specified. This rule does not apply to types used as part of field declarations.

Split the specification into logical units Specifications distributed into several files consisting of small logical units are more accessible using just a text editor then a single large file. Splitting up a specification enables tool builders to omit parts irrelevant for their specific tool.

3 Semantics

This section will describe the meaning of specifications by explaining the effect of declarations.

3.1 A Specification File

```
UNIT :=
  INCLUDE*
  DECLARATION*
```

Specification files are fed into binding generators, which generate code that provides means to deal with instances of the declared types.

SKilL specifications consist of a set of declarations which themselves consist of fields. A declaration is roughly equivalent to a type declaration in an object oriented programming language. The main difference is, that declarations are pure data, because we do not offer a real execution model. The only operations from the perspective of SKilL are loading and storing of data.

A declaration will instruct the language binding generator to create a type which has the declarations name and consists of the fields specified in the body of the declaration. Fields behave just like fields in object oriented programming. Both form the structure of serialized data and are identified using human readable names.

Types are discussed in section 4. Restrictions and Hints are discussed in section 5.1 and 5.2. To do (13)

3.2 Includes

```
INCLUDE :=
  ("include"|"with") <string>+
```

Includes are used to structure a specification into smaller models, e.g. by moving data that is only used by some tools to its own file.

The files referenced by the `include` statement are processed as well. The declarations of all files transitively reachable over `with` statements are collected, before any declaration in any file is evaluated. The order of inclusion is irrelevant. The same file may even be included multiple times by the same `include` statement.

Therefore evaluation of declarations happens as if all declarations were defined in a single file.

3.3 Type Declarations

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

The SKilL specification language is all about type declarations. A type decalartion consists at least of a name and a body, containing field decalartions.

3.3.1 Descriptions

```
DESCRIPTION :=
  COMMENT?
  (RESTRICTION|HINT)*
```

Type (and field) declarations can be enriched with descriptions.

Comments provided in the SKilL specification will be emitted into the generated code[15] to serve as a natural language description of the respective entity. This approach enables users to get tool-tips in an IDE showing him this documentation.

Comments will ignore preceding whitespace and '*' tokens. They end with '*/' and they can not be nested. Comments start with a text and can be followed by tags. Valid tags are

- @see $type/field$
- @deprecated $message$
- @author $name$
- @version $string$
- @note $message$
- @todo $message$

[15] If the target language does not allow for C-Style comments, an appropriate transformation will be applied.

These tags are parsed separately and provided to the back end as a list, to allow the back end to convert them to the format required by the dominating documenting system. The behavior in the presence of other tags is unspecified. Ill-formed tags must not cause a binding generator to fail. Casing of tags is ignored, if tags are followed by a double colon, it is discarded.

If a valid type or field is used as argument to see, the type is translated by the back end to match the generated type or field.

Line breaks and empty lines are dealt with as known by LaTeX: Empty lines separate paragraphs, while ordinary line breaks will be ignored.

The comment system was designed to integrate nicely into doxygen [vH13] or javadoc [jav13].

Restrictions and hints will be explained in section 5.

3.3.2 Subtypes

A subtype of a user type can be declared by appending the keyword `with` (or `:` or `extends`) and the supertypes name to a declaration. A subtype behaves like a subtype in object oriented programming. Subtypes inherit all fields of their super types. Only user defined types can be sub-typed. In order to be well-formed, the subtype relation must remain acyclic and must not contain unknown types.

3.4 Field Declarations

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

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

DATA :=
  "auto"? TYPE <id>
```

Types are sets of named fields. Fields are either constants or real data.

A usual field declaration consists of a type and a field name. In this case, the field declaration behaves like a field declaration in any object oriented programming language, except that the field data will be serialized.

3.4.1 Constants

A `const` field can be used in order to create guards or version numbers. The deserialization mechanism has to report an error if a constant field has an unexpected value. This mechanism is intended to be used basically for preventing from reading arbitrary files and interpreting them as the expected input. The mechanism can be used defensively, because storing constant fields creates a constant overhead and is not influenced by the number of instances of a type.

Only integer types can be used as constants.

3.4.2 Transient Fields

Transient fields, i.e. fields which are used for computation only, can be declared by adding the keyword `auto` in front of the type name. The language binding will create a field with the given type, but the content is transparent to the serialization mechanism. This mechanism can be used to add fields to a data structure, which simplify algorithms computing the interesting data, if these helper fields are not of interest after computation. This is especially useful in combination with the possibility to add or drop some fields while generating the binding for a specific tool. The mechanism can also be used for a field with content that can likely be computed very fast.

The keyword `auto` is used, because the content of the field is computed automatically. Besides the name, it has nothing todo with the **auto** type declaration of C++.

To do (14)

3.4.3 Change-Modifier

Change modifiers can be applied to fields or whole type declarations. Their intended usage is to enable users to specify the contents of their files along their tool-chain by adding (++) data to the specification or by removing (`--`) it. This is an important behaviour in a large tool chain, because it enables tools to have a lean API to the data required by them, without having to maintain a large amount of cloned file specifications.

Usage of ++ and `--` on a type declaration is a shorthand for modifying all contained fields individually. Usage of == is only legal on whole type declarations and is considered to be the "ultimate truth", thus there can be at most one occurrence for each type.

Usage of change modifiers has an impact on the semantics of a specification. Assuming a type *T* has specifications and modifications, then

- There is exactly one type declaration for *T* without a change modifier.
- An arbitrary amount of ++ and – modified declarations with varying restrictions or fields can be applied to a type. A field can either occur with an arbitrary amount of ++ or `--` modifications, but it must not have both.
- There can be at most one == modified type declaration. In that case all other declarations are ignored.

To do (15)

3.5 Field Types

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

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

SETTYPE :=
  "<" GROUNDTYPE ">"
```

```
LISTTYPE :=
  "<" GROUNDTYPE ">"

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

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

Basic types are just an identifier with the type name. For compatibility reasons[16], type names are case insensitive.

Types are explained in-depth in section 4.

3.5.1 Container Types

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 main purpose is to increase the usability of the generated Application Programming Interface (API).

These containers showed to increase the usability and understandability of the resulting code and file format.

3.5.2 Annotations

The `annotation` type is basically a typed pointer to an arbitrary user type. Its main purpose is to provide extension points in the form of references to objects, who's type could not be known at the time of the specification of the annotation field.

3.6 Type Annotations

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

R_ARG := (<float>|<int>|<string>)

HINT := "!" <id>
```

SKilL offers two kinds of type annotations: restrictions and hints. Restrictions can be used to restrict a type, e.g. by reducing the range of possible values of an integer fields to those above 23. Hints can be used to optimize the generated language binding.

Both are explained in-depth in section 5.1 and 5.2.

3.7 Name Resolution

Because SKilL is designed to be downward and upward compatible and offers subtyping, it is possible that a future revision of a file format specification will add a field with a name that already exists in a subtype. In general, it is assumed that the current

[16] Some programming languages, e.g. Ada, do distinguish types by casing of identifiers. Using types which differ only in case in such languages would be very nasty, because type name would have to be escaped in some way. Furthermore, clashes with reserved words can usually solved by capitalization.

maintainer of the super class does not know about all subclasses. Thus, it is desirable to have a mechanism which ensures client code to work correct after such a change.

Therefore, identical field names are legal in SKilL and refer to different fields, as long as they belong to different type declarations.

A generated language binding has to provide means of accessing a field shadowed by a field of the same name in a subtype. In most languages there are built-in mechanisms for this task. Language binding generators shall take care that they do not override field access methods in a way that will actually make the field of the super type inaccessible.

4 The Type System

The description language and the file format are both intended to be *type safe.* The notion of type safety is usually connected to a state transition system. In our context, the only observable state transitions are from the on-disk representation to the in-memory representation and vice versa. Thus, with *type safe* we want to state that deserialization and serialization of data will not change the type of the data. It is further guaranteed that deserialized references will point to objects of the static type of the reference. Further, if one were to deserialize an object of an incompatible type, an error will be raised.

These properties require some form of platform independent type system[17], which is described briefly in this section. The general layout of the type system is visualized in Fig. 1.

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.

4.1 Built-In Types

The type system provides built-in types, which are the building blocks of type declarations.

Integers

Integers come in two flavors, fixed length and variable length. There is currently only a single variable length integer type, namely Variable length 64-bit signed integer (v64). The variable length integer type can store small values in a single byte (see appendix C for details). Large values ($\geq 2^{55}$) and negative values require one additional byte, i.e. nine bytes. To do (16)

To do (17)

To do (18)

[17]In contrast to e.g. C, objects of certain type have a known length and endianness.

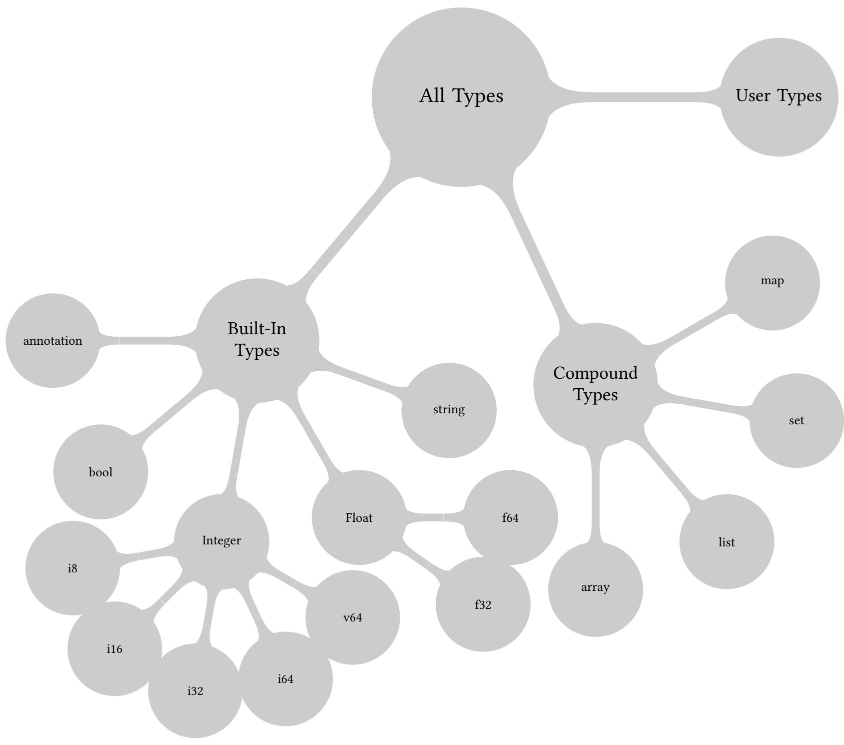

Figure 1: Layout of the Type System

Booleans

Booleans can store the values true($\top$) and false($\perp$). Unlike most C programmers, we do not perceive booleans as integers.

Annotations

Annotations are designed to be the main extension points in a file format. Annotations are basically typed pointers to arbitrary types. This is achieved by adding the type of the pointer to a regular reference. 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 usually result in an error or exception[18]. Therefore language bindings shall provide means of inspecting whether or not the type of the object behind an annotation is known. To do (19)

To do (20)

Strings

To do (21)

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 8. If a language demands NULL-termination in strings, the language binding will ensure this

To do (22)
To do (23)
To do (24)

[18]The reflection mechanism allows for other solutions, but raising an exception is the most obvious reaction.

property.

Strings should not contain NULL characters, because this may cause problems with languages such as C. However, SKilL is NULL character agnostic, thus no guarantees are made.

The API shall behave as if strings were defined with the hint `pure` and `unique` (see section 5.2). To do (25)

NULL Pointer

Fields of type strings, annotations or a user type can store a NULL pointer. These types are by default nullable as NULL pointers are their default values. Fields of these types can be declared nonnull using the nonnull restriction (see Section 5.1.2.1).

Floating Point Numbers

For convenience, it is possible to store 32 bit and 64 bit IEEE-754 floating point numbers. For a description, see [iee08] especially §3.4.

4.2 Compound Types

The language offers several compound types. Sets, Lists and variable length Arrays[19] are basically views onto the same kind of serialized data, i.e. they are a length encoded sequence of instances 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 object twice. All compound types 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 have two or more type arguments.

Note that the serialization format causes containers to have a maximum size of $2^{64}-1$ elements. Thus index types of a container ought to be 64-bit unsigned integers. Language implementers will choose fast over complete, i.e. Scala 2.11 and Java 7 implementations for instance are using 32-bit signed integers, because the underlying JVM will not allow another type to be used. See appendix F for more details. To do (26)

To do (27)
To do (28)
To do (29)

4.3 User Types

User types can be interpreted as sets of type-name-pairs. Built-in types can be wrapped in order to give them special semantics. For example an appointment can be represented as:

Listing 9: Example User Type

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

  /** A topic, such as "team meeting". */
  string topic;

  /** the name of the room. */
```

[19]I.e. arrays, which do not have constant size. Constant length arrays exist as well.

```
  string room;
}
```

To do (30)
To do (31)

4.3.1 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. Field names of super types are not relevant.
- The subtype relation is a partial order[20] and does not contain unknown types.
- Any base type has to be known, i.e. it is a user type defined in any document transitively reachable over include commands.
- The type names must be unique in the context of all[21] types. To do (32)

4.3.2 Equivalence of Type and Field Names

Type and field names, i.e. any strings referenced from reflection data stored in a SKilL file, shall be treated as equivalent, if they were equal after converting all characters to lower case.

We recommend using CamelCase in SKilL definition files in order to provide a hint to the language binding generator on how to separate parts of identifier names. For example an Ada generator will add underscores to the names in the generated interface leading to a more natural feeling for Ada programmers. In order to influence this behaviour, any SKilL specification front-end shall provide an option to interpret a single `'_'` as explicit word separation and double underscores as escape for a single underscore. In this mode, a single underscore will cause a Java generator to use CamelCase.

4.4 Fancy Types

This section will explain fancy types that are made visible to the user, although they do not exist in the binary file per se.

All fancy types can be mapped to equivalent user type definitions.

TBD

4.4.1 Interfaces

TBD To do (33)

To do (34)

4.4.2 Enums

TBD To do (35)

4.5 Views

To do (36)

```
'view' (<id> '::')* <id> '.' <id> 'as'
<type> <id> ';'
```

Views can be used to rename and retype fields of a user type. This feature can be used in two scenarios:

Retypeing is primarily useful if two companion hierarchies exist, which refer each other. Take for examples objects, and types of objects. Abstract objects may refer to abstract types, as their types. More concrete objects, will refer to more concrete types, e.g. functions to function types. There is no sane way of expressing this property besides retyping a *type* field of object types for each sub type to the corresponding sub type in the type hierarchy.

Renaming is primarily useful to deal with historic errors, i.e. with fields that are produced using unacceptable names by tools that can not be modified for what ever reason. Renaming may also be required if inputs from multiple independent tools are merged.

Both renaming and retyping is important to the specification and the generated API only. Nonetheless, an Error must be reported on deserialization in case of infeasible retyping. The related error message shall make clear that the Error is caused by the view onto the data and not the data per se.

Furthermore renaming can be used to move fields down the type hierarchy, if they turn out to be ill-placed. For example a field *name* may seem to be an obvious choice to be placed in a base type. But with continuing growth of a tool chain, sub types may be introduced that do not have obvious names. If this had been known in advance, the name property would have been represented by an interface below the base type that is implemented by all types with natural names. In retrospect, this would not be possible without breaking the file format or implementations of some tools. Thus a specification can be built, that moves the name property to the right places in the generated API only, while keeping the file format and thereby old tools.

An additional benefit of the view concept over TR13 is that renaming allows for complete removal of the rather complicated field name shadowing convention, thus making correct and full implementations of a code generator easier. From now on, field names are unique in the API for any given type.

Note that views can not change restrictions or hints of a field. This feature was dropped, as it would have required introduction of views into the binary file format and thereby would have increased complexity a lot. Views inherit all restrictions and hints of the viewed field except of `!hide` (see §5.2.14).

Views should be used with great care, because they may confuse people, especially if they are working with multiple specifications or language bindings.

Views must not be used in enum declarations.

4.6 Typedefs

To do (37)

```
COMMENT
'typedef' <id> (<restriction>|<hint>)* <id> ';'
```

[20]In fact it forms a forest.

[21]From the perspective of a client, i.e. all types that were declared at the time of the generation of its interfaces plus all types that are ever observed in the form of *unknown* types encoded in SKilL files.

Creates a type name as the restricted and hinted version of another name. This feature is handy in large specifications. Typedefs can be provided with a comment, that is made accessible to the back-end.

Typedefs can syntactially be used to nest containers inside of containers. This is forbidden for coding reasons and will therefore result in an error. For usability reasons using containers in typedefs and typedefs in container definitions is ruled to be legal.

In case of nested containers resulting from typedefs, an error message shall be created at generation time stating that non ground type container argumet resulting from typedef blabla.

Furthermore, typedefs can not be used as super types. This restriction is for readability only.

Typedefs are syntactic sugar which is not exported to the binary file and which shall be projected away to code generators on demand[22].

The front-end shall provide a mechanism to project away all typedefs.

4.7 Default Values

Default values are:

Type	Value
integers	0
floats	0.0
enums	first definition
containers	∅
others	null

4.8 Examples

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

Listing 10: Legal Super Types

```
EncodedString extends string {
  string encoding;
}
```

Error: The built-in type "string" can not be subclassed.

Listing 11: Legal Type Names

```
/**
 The german word for "car".
*/
Auto {
  ...
}
```

Error: "Auto" is a reserved word and can not be used as type name.

Listing 12: Usage of Unknown Types

[22]This is required in order to keep basic implementations at a sane cost.

```
A {
  map<A,B> f
}
```

Error: The field “A.f” refers to a missing type B. Did you forget to include ”B.skill”? To do (38)

To do (39)

5 Type Annotations

SKilL provides two concepts of extending the basic type system used in the serialization process.

The first concept is called restriction and is inspired by the concept of (type or class) invariants. This concept can be used to restrict the set of legal objects storable in a field or the set of legal instances of a class.

The second concept is called hint. Hints are used to improve the generated language binding and do not influence types per se. To do (40)

5.1 Restrictions

Restrictions can be added to type declarations and fields. They can occur in any number at the same places as comments. Restrictions start with an @ followed by a name and optional arguments. If multiple restrictions are used, the conjunction of them forms the invariant, i.e. all of them have to apply. Field and type restrictions do not share a common ID-range and are explained their respective subsections.

If Restrictions are used on compound types, they expand to the components of the respective compound type. Restrictions can not be combined with map-typed fields.

Restrictions are checked by the generated binding at least before serialization and after deserialization. If checking restrictions involves fields, which are not present in a deserialized file, the respective restriction is ruled to hold. This is important to guarantee compatibility with older or newer versions of a file format used in a tool chain. This behavior puts the burden of fulfilling restrictions to the creator of data.

Restrictions are serialized to ensure the asserted properties. The serialization mechanism and an optional recovery strategy is specified alongside each restriction definition. Restrictions with even IDs may be ignored by a binding not implementing the restrictions. These restrictions must implement the specified recovery strategy. Restrictions with even IDs are serialized by the function $[\![ID]\!]_{v64}$[23].

The specification of restrictions is mostly written from the perspective of a user. Nonetheless specifying serialization of restrictions is a necessary lookahead to the next chapters. Please note that although the file is parseable from left to right in one pass, the interpretation of restrictions is not possible until a type block has been parsed completely. This is due to restrictions potentially referring to types that occur later in the block and similar to field data interpretation.

Some restrictions may be skipped by incomplete implementations of SKilL. These restrictions have necessarily no serialized form besides their ID. Furthermore they have even IDs. Relying on these restrictions is nonetheless safe, as a SKilL implementation must treat them as specified, as soon as it treats restrictions of the given type at all. If a SKilL implementation is not implementing a restriction, it has to warn the user about that fact at generation time of the binding. A SKilL implementation

[23]The binary file would not be parsable otherwise.

will always be able to parse binary SKilL files containing restrictions that are either specified in this document or adhere to the criteria of skippable restrictions.

Note that restrictions can be obtained both by reading a file or by adding them to the specification prior to generation of a binding. This chapter contains rules to deal with potential clashes. The usual behaviour is to merge restrictions obtained from both sources and to add them to the resulting file on a write operation or to drop them on an append operation.

Furthermore the expected API behaviour is to provide means of explicitly check restrictions at any given time for a given state. Deferring checking to a languages type system may not be possible, because it may deprive the user of the capability to create an object graph at all. Look for example at the nonnull restriction and think about how to instantiate a tree structure. Although there are solutions to the problem, they usually involve complication of the type hierarchy and the allocation style alike. Restrictions will be checked implicitly after read before revealing the state to the user and before creating an output stream in write/append operations. Thus it is ensured that restrictions hold for binary files and states that ought to be serialized but contradict restrictions can not damage existing data.

Reading a field that does not comply to the specified or serialized restrictions renders the field partial (see section 8.2.6).

5.1.1 Type Restrictions

Type restrictions can be applied to either **any** type definitions or **base** type definitions. They extend to their sub-types automatically.

5.1.1.1 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. Because the combination of unique with sub-typing has counter-intuitive properties, we decided that using the unique restriction together with a type that has sub- or super-types is considered an error, which has to be detected at runtime.

Unique implies an immutable in-memory representation. If the representation were allowed to be mutable, the objects might require a merge after arbitrary operations, which comes at huge runtime costs. Therefore implementations shall provide a builder for unique instances, as well as a means of deleting existing instances upon write operations.

Furthermore, be warned that adding or removing fields from unique types can cause serious compatibility issues.

ID	0
Applies to	any
Serialization	ε
Recovery	unify duplicates and add restriction to output

Listing 13: Examples

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

5.1.1.2 Singleton

There is at most one instance of the declaration. Singletons must not have sub-types. On the other hand, singletons may have super types. In fact instances of enums are represeneted by singleton subtypes of the enum types. This enables enum instances to have fields.

ID	1
Applies to	any
Serialization	ε
Recovery	report an error, if more then one instance exists, otherwise add to output

Listing 14: Examples

```
/** Stores properties of the target system. */
@singleton System {
  string name;
  ...
}
```

Listing 15: Enum like

```
@abstract Weekday {
  string name;
  i8 index;
}
@singleton Monday : Weekday {
}
@singleton Tuesday : Weekday {
}
...
```

is, besides the generated API, equivalent to

Listing 16: Enum like

```
enum Weekday {
  Monday, Tuesday, ...;

  string name;
  i8 index;
}
```

5.1.1.3 Monotone

Instances of this type can not be deleted. Furthermore instances can not be modified once they have received a SKilL ID, because they might have been serialized already. This restriction is basically a type system way of ensuring properties required by fast append operations. It should be used, where ever it is not necessary to delete instances of a type. The monotone restriction can only be added to base types and expands to all sub types of the base type[24].

Monotone types can be treated by a language binding in a very optimized way.

The monotone restrictions implies the monotone hint (see section 5.2.11).

ID	2
Applies to	base
Serialization	ε
Recovery	add to file

Listing 17: Examples

```
/** just kidding */
@Monotone SocialNetworkPost {
  string message;
}

/** this is monotone as well */
PrivatePost extends SocialNetworkPost {
  string recipient;
}
```

5.1.1.4 Abstract

The restricted type must not have static instances. This is similar to the abstract classes in C++ or Java. In contrast to C++/Java abstract, super types can be non-abstract.

Note This restriction does not apply to sub-types.

ID	3
Applies to	base
Serialization	ε
Recovery	raise error, if static instances exist, add to file otherwise

5.1.1.5 Default

Set the default value for a user type. The argument can refer to a singleton type name. The default value is inherited by sub types, but can be changed explicitly for any of them. The default can locally be overwritten by field default restrictions.

[24] This behavior is caused by the fact, that for A <: B, all B instances are As as well. If B would not be monotone, deleting a B would directly delete an A. If A would not be monotone, deleting an A might delete a B.

Note Enums cannot be the target of default restrictions. The default instance of an enum can be changed by moving the desired default value to the top of the list of instances.

ID	5
Applies to	any
Signatures	default(value): $\mathcal{U}$
Serialization	$[\![default]\!]_{TYPE}$
Recovery	replace or add to file

5.1.2 Field Restrictions

Filed restrictions use the type α to denote the fields type. Note that α may even include a type name, if applicable.

5.1.2.1 NonNull

Declares that the argument field cannot be `NULL`.

Note that fields, which have not been initialized, contain `NULL` values if they have no other default specified explicitly.

Cannot be used on `annotation`.

ID	0
Applies to	String, Usertypes
Signatures	`nullable`:
Serialization	ε
Recovery	add to file

Listing 18: Examples

```
Node {
  /* null-edges are pointless */
  @nonnull Node[] edges;
}
```

5.1.2.2 Default

To do (41)

integer und floats sind ihre entsprechenden werte, solange sie in Range liegen. strings können string literale sein pointer artige können namen von Singletons eines passenden typs sein.

To do (42)

To do (43)

To do (44)

ID	1
Applies to	ground types
Signatures	`default(value)`: α
Serialization	$[\![value]\!]_{\alpha}$
Recovery	add to file; replace serialized default, if one exists; raise error if default value is invalid[25].

Listing 19: Examples

```
/** a multi-graph with a weighted edges and labeled
    nodes. */
```

```
Graph {
  Node[] nodes;
  Edges[] edges;
}

Node {
  @default("")
  string label;
}

Edge {
  Node from;
  Node to;

  @default(1.0)
  f32 weight;
}
```

5.1.2.3 Range

Range restrictions are used to restrict ranges of integers and floats. They can restrict the minimum or maximum value or both. Restrictions can be inclusive or exclusive – the default is inclusive.

Note that this will change the implicit default value of the argument field to *min* iff $0 \notin [min, max]$. If an explicit default value has been specified, it must not contradict the range.

ID	3
Applies to	Integer, Float
Signatures	`range(min, max, boundaries)`: $\alpha \times \alpha \times string^?$ `min(min, boundaries)`: $\alpha \times string^?$ `max(max, boundaries)`: $\alpha \times string^?$
Serialization	$[\![min_{inclusive}]\!]_\alpha \circ [\![max_{inclusive}]\!]_\alpha$
Recovery	add to file; intersect with serialized range; raise an error, if intersection is empty

Listing 20: Examples

```
RangeRestricted {
  @min(0)
  v64 natural;

  @min(1)
  v64 positive;
  /* or */
  @min(0,"exclusive")
  v64 positiveAlt;

  @range(0.0, 360.0, "inclusive,␣exclusive")
  f32 angle;
}
```

5.1.2.4 Coding

The arguments field data chunks are encoded using the argument coding. No codings are specified by SKilL. Potential arguments include ”zip” or ”lzma” or some sort of encrypting container. To do (??)

Usage of codings is discouraged, because of the burden it puts on implementing a binding generator for some combinations of programming language and platform and the implied loss of portability.

Usage of coding might be necessary to fulfill legal obligations in a specific use case. To do (46)

Note Coding is designed in a way, that allows skipping fields if the coding is unsupperted. If a coding requires additional information, this information has to be placed somewhere in the field data or in a side-channel of the application, such as another file or a client-server communication.

ID	5
Applies to	any
Signatures	coding(name): $string$
Serialization	$[\![name]\!]_{string}$
Recovery	add to file on write; if unsupported, treat field ignore-hinted and the type as monotone-restricted

Listing 21: Coding Examples

```
ToolDescription {

  /** the log is written by a tool, but usually unused
     afterwards */
  !lazy
  @coding(”zip”)
  string log;
}
```

5.1.2.5 Constant Length Pointer

To do (47)

The argument pointer is serialized using i64 instead of v64. This can be used on regular references and annotations. This can be combined with the coding restriction. The restriction makes only sense if the generated binding supports lazy reading of partial storage pools and if the files that have to be dealt with, would not fit into the main memory of the target machine. Using this restriction will most certainly increase the file size and does not restrict any pointer targets.

This restriction is serializable and, thus, does not affect compatibility in any way.

ID	7
Applies to	String, Annotation, Usertypes
Signatures	constantLengthPointer
Serialization	ε
Recovery	add to file on write; if unsupported, treat field ignore-hinted and the type as monotone-restricted; on append, use whatever the file specified.

Listing 22: Examples

```
/* stored points to information may exceed the available
   main memory, thus we have to access it directly from
   disk */
PointsToTargets {
  @constantLengthPointer
  Context context;
  @constantLengthPointer
  HeapObject object;
  @constantLengthPointer
  PointsToSet targets;
}
```

5.1.2.6 OneOf

OneOf restrictions are used to restrict potential targets of annotations. Restrictions are still nullable. This is the foundation for tagged union types.

ID	9
Applies to	Annotation, Usertypes
Signatures	`oneOf(type1, ..., typeN)`:$\mathcal{U} \times \cdots \times \mathcal{U}$
Serialization	$[\![type_1, \cdots, type_n]\!]_{TYPE[]}$
Recovery	add to file; intersect with serialized range; raise an error, if intersection is empty

5.2 Hints

Hints are annotations that start with a single `!` and are followed by a hint name. Hints are used to optimize the behavior of the generated language binding. They do not impact the semantics of type declarations or stored data. Therefore they will not be serialized.

Language bindings shall provide the same public interface as if no hints were used.

Language binding generators shall provide an option that adds a hint to all applicable declarations.

5.2.1 Owner

Can be used on: Base type declarations.

Arguments: Names of tools that own the type hierarchy, i.e. that can modify instances of the annotated base type.

Note: At generation time, the binding generator needs to know tool names, that are owned by the generated binding, in order to hard-link this behavior.

Note: Arguments are lists of names, in order to allow to separate the forest of types into different areas of concern.

Types without this hint are owned by every binding. If a binding is created without ownership information, it owns all types implicitly. These two rules are required in order to keep ownership optional, even if annotations exist.

Note on implementation: All types that are not owned by the tool are marked as ReadOnly.

Listing 23: Owner Example

```
!owner(ColorOwner)
Color { i8 r; i8 g; i8 b; }

/**
A GraphProvider should be a ColorProvider as well, but
    if it's not, it can still set existing
colors. A ColorProvider on the other hand is able to
    create new colored graphs.
*/
!owner(ColorOwner, GraphOwner)
Node {
  Color color;
  set<Node> edges;
}
```

5.2.2 Provider

Can be used on: Type or field declarations.

Arguments: Names of the tools that provide the target field.

Note: If used on type declarations, it extends to all field declaration inside of it and all sub-types.

Note: If used on a field declaration, which belongs to a type declaration, that is not provided by the same provider, the provider of the surrounding type does not provide the field.

Note: A Provider name has to be specified at tool-generation time.

The target field can not be rendered partial by a write or append operation.

Listing 24: Provider Example

```
!provider(ColorProvider)
Color { i8 r; i8 g; i8 b; }

/**
A GraphProvider creates graphs.
A ColorProvider adds color.
*/
!provider(GraphProvider)
Node {
  !provider(ColorProvider)
  Color color;
  set<Node> edges;
}
```

5.2.3 Remove Restrictions

Can be used on: Anywhere

Arguments: Optional restriction names; if none is supplied, all restrictions are removed; if "unknown" is specified, only restrictions unknown to the generated binding are removed

Removes restrictions obtained from a file instead of keeping and adhearing to them. If a restriction kind is to be removed but the same kind is specified, the specified kind will be present in the output. This behaviour is required in order to replace some restrictions with arguments, such as range, because their recovery strategy may have undesired consequences.

Rationale: This Restriction is used to deal with decisions made in the past. This behavior is usually undesired, therefore it has to be enabled explicitly.

Listing 25: Remove Unknown Restriction Example

```
/** in the past, there was only one display, i.e. we
    want to remove the @singleton */
!removeUnknownRestriction
Display {  }

/** From now on, we allow to specify a dedicated primary
    display */
@Singleton
PrimaryDisplay{
  Display current;
}
```

5.2.4 Constant Mutator

The restriction can be used to change a constant. This might be required in order to upgrade the version of a data set.

The file reader will accept values between *min* and *max* and write a new file with value *new*.

Note Appending is not possible if a constant changed. A binding shall silently fall-back to write.

Listing 26: Constant Mutator Example

```
/** From now on, we allow to specify a dedicated primary
    display */
@Singleton
ToolInfo{
  /** the guard stays the same */
  const i16 guard = 0xABCD;
  /** we accept files adhering to version 1 as well,
     because the format is compatible in one direction */
  !constantMutator(1, 1)
  const i16 version = 2;

  string toolName;
}
```

5.2.5 Mixin

The actual Implementation of UserTypes is left to the client. This is a very generator dependent feature. To do (48)

5.2.6 Flat

Can be used on a type definition with a single field to indicate that the type definition shall be hidden in the exported API. This is necessary, if containers ought to be shared between multiple entities.

5.2.7 Unique

Can be used on: Type declarations.

Serialization shall unify objects with exactly the same serialized form. In combination with the @unique restriction, no error shall be reported on serialization. To do (49)

Note that this will increase the runtime complexity of the serialization phase from $O(n)$ to $O(u \cdot nlog(n))$, where n is the size of the state and u is the number of types declared to be unique.

5.2.8 Pure

Can be used on: Type declarations.

Deserialized objects of the annotated type shall not be modifiable. The generated interface will provide a copy operation, which will create a modifiable copy of the object. An example of this behavior is the string pool. An equivalent, in terms of API observable behavior, would look as follows:

Listing 27: User Strings

```
!pure
!unique
UserString {
  !OnDemand
  i8[] utf8Chars;
}
```

5.2.9 Distributed

Can be used on: Field declarations.

A static map will be used instead of fields to represent fields of definitions in memory. This is usually an optimization if a definition has a lot of fields, but most use cases To do (50) 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[26] and the access time for the given field and will only be a benefit for memory-cache locality reasons, because single objects can be significantly smaller[27]. The internal representation will change from `o.f`, i.e. a regular field, to `pool.f[o]`, 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[28].

5.2.10 OnDemand

Can be used on: Field declarations.

Deserialize the respective field only if it is actually used. OnDemand implies distributed. This hint should be used, if fields are usually not accessed, e.g. in the context of error reporting.

5.2.11 Monotone

Can be used on: Base Type declarations.

Instances of the argument type will not be deleted. New instances can be added to the state. This allows an optimized treatment of data, because assumptions about object IDs can be made.

Accessing lazy fields of instances of monotone types is about as efficient as accessing ordinary fields. Note that usage of v64 and references requires deserialization of all fields in a type block. Thus, it might be worth considering usage of constant length integers or the constantLengthPointer restriction, if this case is encountered on a regular basis.

In contrast to the monotone restriction, the hinted monotone property applies only to the generated binding.

5.2.12 ReadOnly

Can be used on: Base Type declarations.

The generated code is unable to modify instances of the respective type. This hint can be used to provide a consistent API while preventing from logical errors, such as modifying data from a previous stage of computation. The ReadOnly hint expands to all subtypes of the base type, because this is the only safe way[29].

ReadOnly implies Monotone. If a ReadOnly field is partial, an error has to be thrown while reading the file.

[26] Because additional data structures, such as trees, are required in order to provide acceptable access times.

[27] Note this does not apply to operations on distributed fields, but to operations on objects having distributed fields.

[28]As a matter of fact, this is not true in case of non object oriented languages such as C.

[29]This behavior is caused by the fact, that for A <: B, all B instances are As as well. If A would not be ReadOnly, modifying an A would directly modify a B. If B would not be ReadOnly, modifying a B might modifying an A.

5.2.13 Ignore

Can be used on: Type and Field declarations.

The generated code is unable to access the respective field or any field of the type of the target declaration. A language binding shall raise an error (or exception), if the field is accessed nonetheless. This hint can be used 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. This is actually more restrictive than deleting fields from declarations, because the generated reflective API will respect this hint.

Ignore overrides any other hint and a binding may ignore any restriction applying to an ignored field.

5.2.14 Hide

Can be used on: Field declarations.

The generated code will not produce an API for the field declaration. Hidden fields can be made accessible by views.

5.2.15 Pragma

Can be used on: anything.

Usage: `!pragma <ID>+`

Example: `!pragma tree`

The pragma is used to pass additional information to the code generator. This mechanism is intended to be a short term solution to ease development of the language.

A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.

Douglas Adams in *Mostly Harmless*

Part II
User Interface

To do (51)

To do (52)

To do (53)

6 In Memory Representation

This section is about the generated API provided to the programmer and the representation of objects inside memory. It is meant as a concise guideline for implementers of language binding generators.

The behaviour in case of a mismatch between the expected type of a field and the type stored in a file is left unspecified. This will be changed in a future version. For now only matching types can be assumed to be compatible and portable.

To do (54)

To do (55)

To do (56)

To do (57)

6.1 API

The generated API has to be designed in a way that integrates nicely with the language's programming paradigms. For example 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 from the state object. The serialized types can be represented by interfaces providing getters and setters, using hidden implementations only visible for the state object.

To do (58)

To do (59)

To do (60)

To do (61)

To do (62)

To do (63)

To do (64)

To do (65)

6.2 Abstract State and State Transitions

Because we do not want to talk about each kind of state transition or a state individually, we provide a small abstraction. We will call a state σ. A state knows all types τ that it contains. Each type τ knows its fields and its instances. We will call a binary skill file sf. Entities used in this section will be treated as if they were ordered sets.

We will talk about state transitions for create, read, modify, write, append, order and check.

create: $() \to \sigma$ This transition creates a new σ with types that perfectly match the specification used to generate the binding. The new σ contains no instances except for singleton types.

read: $sf \to \sigma, \lightning$ This transition creates a new σ and ensures that $\forall \tau \in sf, i \in \tau.\tau \in \sigma \wedge i \in tau$. Note that this specification enables implementations that behave as if they had read the data from the file until the user is actually requesting it. Conformity between the type system of sf and the specification has to be checked before producing an observable σ. A successful read is guaranteed to pass a check immediately after, thus $\forall sf.read(sf) = \lightning \vee check(read(sf))$. If sf is empty, then $read(sf) = create()$.

modify: $\sigma \to \sigma'$ Modifications are what ever a user does with a state. This transition is not constrained in any way.

write: $\sigma \to sf$ If $check()$ then $read(write()) = .$ Otherwise a write transition is illegal. Note that although $write \circ read = id$, $read \circ write \neq id$ in general, because the binary representation has some degrees of freedom that can not be observed in memory and the read operation may change the type system of σ compared to the one obtained from file.

append: $(sf, \sigma) \to sf$ If $check()$ and $\forall \tau \in sf, i \in \tau, \tau' \in \sigma, i' \in tau.i \subseteq i'$, where τ' is the in memory representation of τ and i' is the in memory representation of i, i.e. the in memory representation of σ is a perfect match for all information stored in sf then $read(append(sf, \sigma)) = \sigma$. Note that σ can only contain new types, larger types or new instances.

order: $\sigma \to \sigma'$ Changes the order of types and instances in a state. The transition ensures that $check(\sigma) = check(order(\sigma))$, $order(\sigma) = order(order(\sigma))$. Furthermore for all σ, σ' that are equal if compared neglecting their order $write(order(\sigma)) = write(order(\sigma'))$.

check: $\sigma \to \top, \bot$ True iff all restrictions in σ are satisfied.

6.3 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 can do the job in a sufficiently efficient way. 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. To do (66)

To do (67)

6.3.1 Proposed Data Structures

A state has to contain at least

- an array of strings
- type information obtained from a file
- storage pools

To do (68)

A storage pool has to hold the images of (or references to) 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. For example, 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 the case, because the pool holds the field data.

Now that we have a representation of objects, we still have to organize storage of objects across storage pools. The possibility of inheritance requires a view onto stored objects, which looks as if they were stored in multiple pools at the same time. We propose to store supertype and subtype information inside pools as references to the respective pools. The objects should be stored as a "double linked array list"[30], 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 instances of a type uses an iterator over all instances of a pool in combination with iterators over the pool and all its subpools. Therefore creating and deleting objects is an amortized O(1) operation[31] 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 the mere presence of distributed fields will change the worst case runtime complexity of these operations from (amortized) $O(1)$ to $O(log(n))$, where n is the largest number of objects with a common distributed field in a state.

6.3.2 Reading Unmodified Data

Reading unmodified data is basically done by creating objects with ascending IDs while processing all eagerly processed fields. Pointer resolution in an unmodified state is an O(t) operation, where t is the number of subtypes of the static type of the pointer[32].

During the reconstruction of the initial dataset, an array in the base pool may be used to reduce the cost to O(1)[33]. However, this helper array has to be dropped, as soon as the base pool is modified.

6.3.3 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 might invalidate any existing object ID, i.e. deleting objects or inserting objects into non-empty 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 lurks in an yet untreated field.

[30] In Java this would be a LinkedList<ArrayList< ? extends B > >, where B is the provided interface of the base type of a pool. Note however, that the LinkedList-part is expected to be realized using references between storage pools.

[31] This can be achieved by appending new objects to the end of the pool and deleting existing objects by creating holes inside the pool. The compaction of the storage pool is done while writing the output and can be amortized into the writing costs.

[32]This is usually a very small constant.

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

6.3.4 Reading Data in a Modified State

Reading Data from disk 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 invariant 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 is expected to eliminate the additional cost, because transformation of stored state is often the subject of monotonic growth, because each step of a computation usually adds instances of a type, which had no instances in a previous step.

6.3.5 Writing Data

Data which is lazy and modified can not directly be written to disk, because 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 writing is not per se a performance issue.

6.3.6 Final Thoughts on Runtime Complexity

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

Reading data without modifying is in $O(in)$, even if only a part of the data is read[34]. 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 (or seeking through) 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 for performing the modifications.

Only the usage of a lot of non-monotonic 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. On the other hand, lazy fields can be very valuable, if their data is used for error reporting or debugging. To do (69)

[34]Note that a file can contain little payload compared to the type information stored in the file, therefore $O(in)$ is a sharp complexity estimate.

Omnia mutantur, nihil interit.
(All things change, nothing perishes.)

Ovid in *Metamorphoses, XV. 165*

Part III
Binary File Format

7 Impact of changing Specifications

Most people that have to provide the specification of a file format tend to think that they know the format of that file. This is a fundamental problem and this section will briefly explain why one can not know the exact specification of any given file format.

Let us think about a very simple tool description:

Listing 28: ToolDescription.skill

```
Tool {
  string name;
}
```

This description can be used by a tool chain driver to identify the tools that have been used to produce the content of a file. In fact this is useful information in a pre-internet world. In modern times, tools can be obtained over the internet, thus adding e.g. the location of a repository containing the tool can be used by a tool chain driver to run tools, that have not been installed on the target machine before. Thus, a modern feature may turn the specification into:

Listing 29: ModernToolDescription.skill

```
URL {
  ...
}

Tool {
  string name;
  /* not null, if a public repository containing the
     tool exists */
  URL repository;
}
```

Please note that changing the specification did not render old files incompatible to the new format, thus no data is lost! After some experiments with the new format, one might have observed, that the driver can execute tools which are not installed. This can be used to provide a file using a similar format that contains all known tools. The file can just be called `knownTools.sf` and can be processed by any tool that is able to read the tool info as described above.

The means to deal with format changes that are provided by SKilL are carefully designed. They are the result of examination of changes in the Bauhaus tool chain[RVP06], which by now is 20 years old and consists of over 120 tools.

An immediate consequence, besides many rules that deal with change, is that types and fields have names, which are only used to map them to expected types. For the sake of efficiency, those names are not used to link data together. Linkage is achieved by indices only.

8 Serialization

To do (70)
To do (71)

This section is about representing objects as a sequence of bytes. We will call this sequence *stream*, its formal type will be called S. We will assume that there is an implicit conversion between fixed sized integers[35] and streams. We also make use of a stream concatenation operator $\circ : S \times S \rightarrow S$.

We will use upper case letters for types (e.g. A) and lower case for instances of the respective type (e.g. a). The type $\mathcal{T}$ denotes the set of types and τ a type. We will use τ_i for arbitrary type names and f_i for arbitrary fields, i.e. $\tau_i.f_j$ is the j'th field of the i'th type.

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, $[\![\_]\!]$) explained below can be easily computed.

The serialization function $[\![\_]\!]_\tau : \tau \times \mathcal{T} \rightarrow S$ maps an object _ of a type τ to a stream. Usually the type of the object can be inferred from context, thus we can simply write $[\![\_]\!]$. During the process of (de-)serialization, the type of an object can always be inferred from context. Note that the definition is given as a set of equations. Therefore the specification of serialization and deserialization are identical. To do (72)

To do (73)
To do (74)
To do (75)

8.1 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. Objects are organized into their storage pools, i.e. the index function is calculated.
    - If the state was created by deserialization and indices have changed[36], fields using these indices have to be updated.
    - All known restrictions have to be checked.
3. The output stream is created as described below.

8.2 General File Layout

The file layout is optimized for fast appending of new objects. It is further optimized for lazy and partial loading of existing objects. It does also support type-safe and consistent treatment of unknown data structures. In order to achieve these goals, 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. We want to be able to diagnose file corruption as early as possible, therefore most information stored in a file relies only on information that has already been processed. The only exception to this rule are field types referring to user types which are declared later in the same block.

Therefore the file is structured as an altering sequence of string blocks and type blocks, starting with a string block. The layout of string blocks (S) and type blocks (T) is visualized in figure 2, details will be explained in the following sub-sections. To do (76)

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

[36]Indices can change by inserting objects before an existing object or by deleting objects. This is caused by the base pool index concept explained in section 8.3

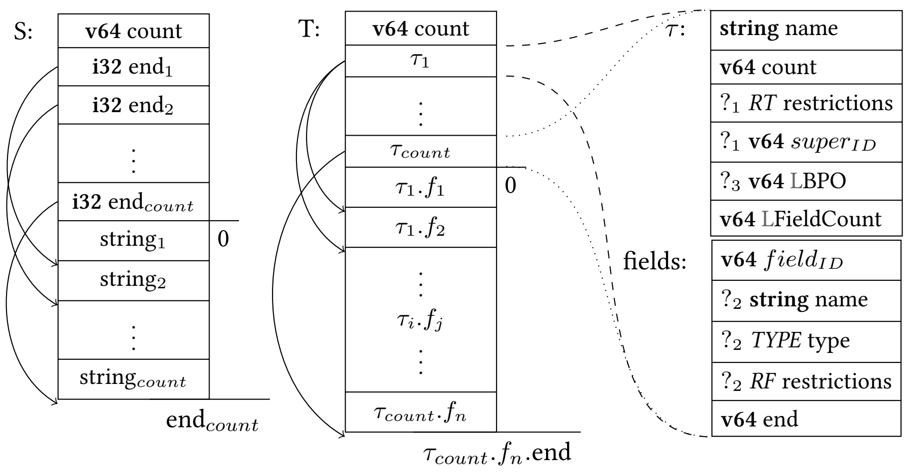

Figure 2: Visualization of the layout of string(S) and type(T) blocks. Type descriptors (τ) and their fields can drop fields in some contexts ($?_i$). Fields prefixed with a gray L contain information only relevant for the local block. The data chunks of blocks start at the respective **0** and reach until the respective **end**. Arrows indicate end-offsets – type blocks have one per field declaration. Field restrictions have to be placed after the fields type, because their translation may depend on the type. LFieldCount many field descriptions are located after LFieldCount – similar to the serialization of arrays.

8.2.1 Layout of a String Block

A string block starts with a v64 *count*, which stores the number of strings stored in the block. It is followed by *count* many i32 values, which store the offset[37] of the end of the respective string. The stored offsets split the data following after the last offset into utf-8 encoded strings.

The individual strings can be decoded using their index and the previous index (or 0, if there is no previous index). The strings stored in the string block are used by the following type blocks. For efficiency[38] reasons, strings used as type or field names shall be stored in lower case only. If a string containing user data equals a type/field name with different casing, a copy has to be made.

If two deserialized strings have the same in memory representation, only one copy shall be stored, i.e. strings behave as if they were unique without the requirement of deserializing strings that have not been used. This relaxation still ensures that types refer to unique strings, as all type/field names have to be parsed, but the interference with potentially many user strings wont hurt you on store operations.

The serialization function of a string block can be summarized as:
$S(s_1, \ldots, s_n) = [\![n]\!]_{v64}[\![end_1]\!]_{i32} \cdots [\![end_n]\!]_{i32}[\![s_1]\!]_{utf8[]} \cdots [\![s_n]\!]_{utf8[]}$

8.2.2 Layout of a Type Block

The instances of all types are organized into storage pools (see section 8.3), which are stored in type blocks. A type block starts with a v64 *count*, which stores the number of *instantiated* types stored in the block. The *count* is followed by the respective amount

[37]in Bytes
[38] The lower case conversion during deserialization is optimized away.

of type declarations.

Type declarations themselves contain field declarations, which contain end-offsets into a data chunk located at the end of the type block, i.e. the field data is stored between the end-offset (or the start of the data chunk) of the previous field and the end-offset of this field. The Local Base Pool Offset (LBPO) field (marked with $?_3$ in Fig. 2) is only present if there is a super type and count is non-zero. The LBPO is used to indicate which fields in a supertype declaration belong to instances of the current type. The local version of the Base Pool Offset (BPO) works in the same way as the BPO in storage pools (see arrows in Fig. 5). The LBPO field is only present if there is a supertype and the type has not yet been defined in a previous type block. Otherwise it is not needed and can be assumed to be 0.

A type is *instantiated*, if the block adds new instances, fields or both. For example, the first block may add a type *node*, with an *ID* field. The second block, which is the result of a graph coloring tool, adds a *color* field to the *node*. We will come back to this example in section 8.2.8.

Note Type blocks are designed carefully to ensure that strings referring to type or field names have to be compared with other strings exactly once in the process of linking a type definition to expected types. After that, the (implicitly present) ID of types and fields is used, simplifying and fastening operations on types and fields.

8.2.3 Type Order

Type order is a partial order that linearises the type hierarchy. It ensures that all data required for the interpretation of a type definition inside a type block is already present. This causes an improvement in both, error reporting and deserialization speed, as well as simplicity of the generated binding.

The partial type order is the guarantee, that a super type is smaller, then a sub type. If append operations to a file are performed in ascending partial type order, each block of the file will necessarily be in partial type order, even if specifications of tools contributing to that file differ.

A stable type order is obtained from adding lexical order to the unconstrained cases of type order. Stable type order provides a faint chance of binary equivalence of two files with equivalent contents. The stable variant is provided by the front-end and will increase compatibility of generated code with version control systems as regenerating code will not change the result if the specification did not change.

If the knowledge about the type hierarchy differs between producers and the consumer of the type hierarchy, the file will only be partially type ordered in general. Thus a file reader can only assume that the input is in partial type order.

8.2.4 Effects of On Demand Deserialization

In this context, *on demand* deserialization means "the ability to skip data".

Inside of string blocks, most indices and string payload can be skipped. The offset type is `i32`, which allows for random access deserialization of individual strings, if the ID is known. This feature is very valuable, if there are many user data strings and few type names. The decision has also the consequence, that strings can only have 2^{32} bytes of data. We consider this limit irrelevant, because a user can still use a byte array to represent string-like objects exceeding this limit. Inside of type blocks, all type information has to be processed in any case. The field data can be skipped completely.

Thus the laziest processing of a SKilL file will read the `count` fields of blocks, the last index of string blocks and all type information, including strings storing type and field names. Even in case of large files with many types, the amount of processed data is expected to be below a megabyte. To do (77)

8.2.5 Effects of Appending

The desire to append new data of an arbitrary kind to an existing file, without having to rewrite existing data, affects mostly the hidden part of the generated language binding. From the perspective of the file format, appending adds altering sequence of blocks, instead of a single string block followed by a single type block. Means to add fields or instances to existing storage pools (see section 8.3) are made necessary by this feature as well. The omitted data is marked with $?_1$ and $?_2$ in Fig. 2: Fields marked with $?_1$ do only appear in the first block and are left away in all other blocks adding data of the respective type. Fields marked with $?_2$ are only present if the field is added to a type. This ensures that later extensions of a type can not contradict its definition in an earlier block.

If appending is not used at all, the overhead, compared with a similar format that does not allow for appending, is about two bytes. If appending is used in a way, that only adds new fields or new types, the overhead is still in the range of several bytes. Adding instances to existing storage pools is expected to create an average overhead of about forty bytes, mostly caused by end-offsets of added field data.

If additional instances are added, the order of already instantiated fields is the order of their occurrence in previous blocks. If additional instances are added in combination with adding new fields, the new fields are located after the existing fields. The already existing fields contain data for the new instances, whereas new fields contain data for all existing instances.

8.2.6 Partial Fields

To do (78)

A field that lacks instance data or does not comply to its restrictions is called *partial*. If ownership is used, only tools owning the partial field, i.e. the surrounding type-tree, are allowed to access and modify the field. It is the duty of a tool that is either owner or provider of a field to turn a partial field into a total field. Thus, an error shall be produced, if an owned type contains a partial field on a write operation. This shall happen even if the field was partial in the source file.

Fields with a default restriction will only be rendered partial if they do not comply to a restriction. Default values are serialized alongside regular field data. Therefore, in this case *missing* field data can be substituted with default values.

8.2.7 Field IDs

The ability to add field data only for some fields requires the serialization format to treat fields by IDs instead of a positional approach. Field IDs are counted from 1, because 0 is reserved for a runtime representation of SKilL IDs. Field IDs are dense, and fields appear always in an ascending order. Field IDs are only relevant to the serialization mechanism and do not correspond to a specification in general.

8.2.8 Node Example

To do (79)

Let us assume two tools with two SKilL specification files:

Listing 30: Specification of the Node Producer Tool

```
Node {
  i8 ID;
}
```

Listing 31: Specification of the Node Color Tool

```
Node {
  i8 ID;
  /** for the sake of simplicity a string like "red" */
  string color;
}
```

Let us assume, the first tool produces two nodes (”23” and ”42”) and the second appends color fields to these nodes (”red” and ”black”). The layout of the resulting file, after the second tool is done, is given in figure 3. Actual field data is kept abstract, because serialization of field data will be explained below. The data produced by the first tool is S and T, the second tool produces S’ and T’.

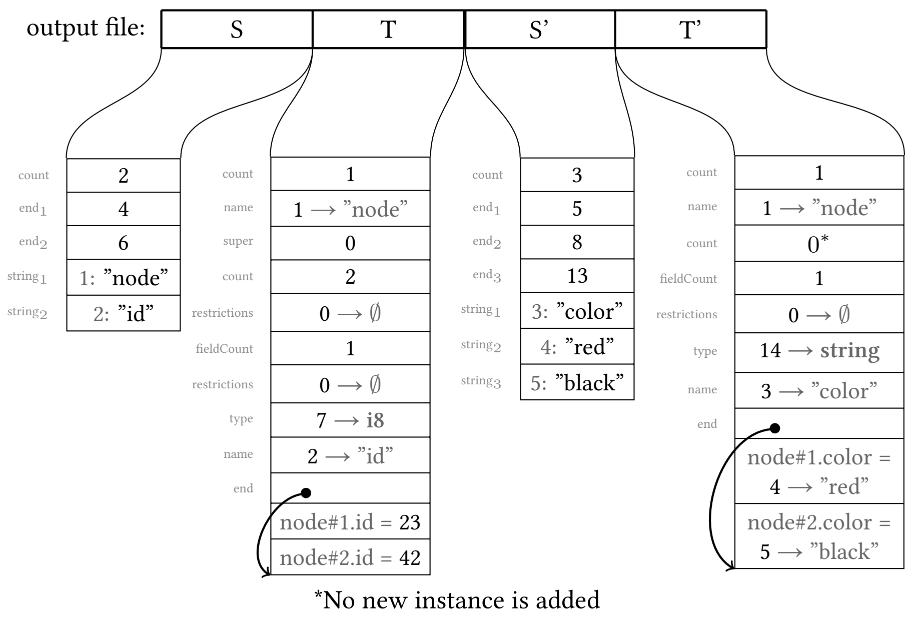

Figure 3: Illustration of the file obtained after running the example tool chain. Gray parts are the interpretations of the respective value and will be explained below. Light gray annotations shall serve as a reminder of the meaning of the contents of the respective field. In section 8.6 there is a more complete example.

We will see in the next section, that fields referring to other objects are stored by reference. Strings are somewhat a special case of this. Strings are used both, as a first class type and to represent type information itself. This has the consequence, that string pools contain user data and data, which might not be directly observed by a user.

Adding instances

Now let us assume, that the first tool is ran twice, adding (”23” and ”42”) in the first run and (”-1” and ”2”) in the second; we wont run the coloring tool, thus S/T are the result of the first run and S*/T* are the result of the second run.

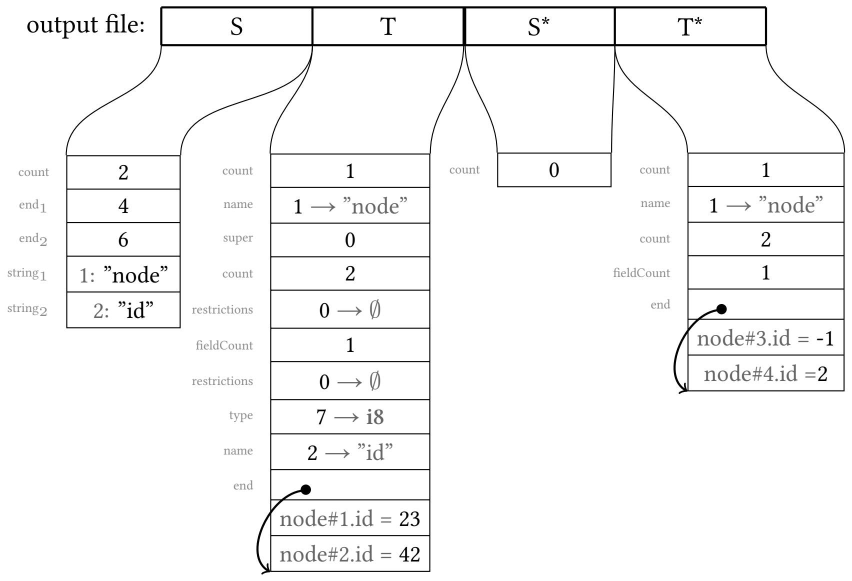

Figure 4: Illustration of the file obtained after running the example tool chain. Gray parts are the interpretations of the respective value and will be explained below. Light gray annotations provide contents descriptions. Note that string blocks (and type blocks) can be empty, but must not be omitted.

8.3 Storage Pools

This section contains the serialization function for an individual storage pool. A storage pool stores the instances of a type, i.e. all field data of the type of the pool. For now, we assume that storage pools are not empty.

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 baseTypeName is either the index of the string used as type name of the base type or 0x00. The typeName is the index of the string used as type name of the argument type. The index is the unique index of the argument object. Indices are assigned ascending from 1 for instances of a base type, for each base type. For example the index 23 can be given to a *Node* object and a string ”hello”, because they do not have a common base type. Indices do not have holes. Although indices are serialized using the v64 type, they are treated as if they were unsigned integers[39]

[39]That is the index -1 refers to object number $2^{64} - 1$.

A basic concept of the serialization format is to store the data grouped by type into storage pools. This concept enables us to obtain type information of objects from their position in a file, instead of storing a type descriptor with each object.

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 knows how many instances it holds. Storage pools with a supertype store the name of the supertype and a BPO. Further, we assume that objects, which are written to a file, have indices such that for any type, all instances of that type have adjecent indices.

A short example (Fig. 5) illustrates the concept. It contains five types **A**, **B**, **C**, **D**, and **N**. For the sake of simplicity, each type has a single field of an arbitrary type τ_x (serialization of field data will be explained in the next section). **A** and **N** are base types. **A** has 6 instances. **B**, **C** are subtypes of **A**; **D** is a subtype of **B**. **B**/**C**/**D** have 4/1/1 instances and BPOs of 2/6/5, respectively. The arrows represent the end-offsets stored in the field descriptions, which are used to separate the data part of the type block into field data. The *index*-row displays the index of the object that belongs to the serialized field. To do (80)

For the sake of readability, the header of the type pool is omitted in the stream part of the picture.

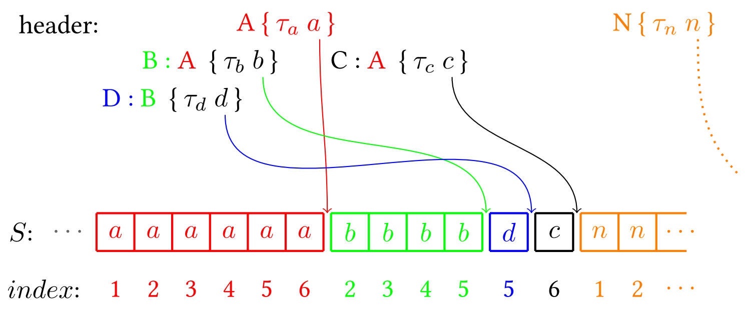

Figure 5: Field data of several pools stored in a type block. Arrows indicate the values of data fields of the respective field description in the header of the type block. S shows a part of the data chunk of a type block. LBPOs of B/C/D are 1/5/4.

Although illustrations above might suggest an order of field data, this is in fact not the case. This allows for optimization during serialization, such as multi-threaded encoding of different fields at the same time.

8.3.1 Subtypes

TBW To do (81)

Invariants of BPOs: $first.index - base.first.index = bpo$ $first.index + count = last.index$ if there is any subtype exists, then there must be a subtype with $count = sub.bpo + sub.count$ and for each subtype, the following equation holds: $count \leq sub.bpo + sub.count$

8.3.2 Appending Fields to existing Pools

Appending a field to an existing pool is rather easy, because existing objects are not directly modified. The respective type block will have a type declaration with a field declaration of the added field. The added field data is stored in the data part of the type block.

The order in which existing fields are appended to a file is unconstrained, because all required information can be obtained by an inexpensive lookup.

8.3.3 Appending Objects to existing Pools

To do (82)

If objects are added to existing pools, all type pools (i.e. the pools of the type and all its supertypes) have to be updated. If a supertype exists, the local base pool start index will give the start index of the added objects in the local pool. With the types as in figure 5 we can for example add several object in three blocks.

Let T1 contain 6 objects of types `aabbbc`, T2 contain 4 objects of types `bbdd` and T3 another 3 objects `acd`.

T1 contains the full type information of types A,B and C. There are 6 A instances, 3 B instances (with LBPO=3) and 1 C instance (with LBPO=6).

T2 contains only the type name of A and B (LBPO=1) and field data for 4 instances each. Additinally, there is a new type D (with LBPO=3) and D-field data for two instances.

T3 contains field data for A, B (LBPO=3), C (LBPO=2) and D (LBPO=3) objects.

The in-memory representation after loading these objects is expected[40] to be either `aabbbcbbddacd` or `aaabbbbbdddcc`, depending on used hints and restrictions . To do (83)

8.3.4 Omission of Data

The serialization of an empty storage pool is its name and super information, with all other values set to zero. This behaviour ensures the integrity of the written file while minimizing size at the same time.

Fields that use only default values, may choose not to serialize any field data. Fields to user types that contain null-Pointers only, may not be serialized at all.

Storage pools that are both empty and not used referred to by a non-empty storage pool, may not not be serialized at all.

The rules above address an issue of a prior version of SKilL where significant amount of type specification had to be stored in seemingly empty files. Simply omitting types that have no instances is not enough, because other types may refer to them. Furthermore the default restriction mechanism is used to further compress files that have no entropy per se.

8.4 Serialization of Field Data

In this section, we want to describe the serialization of individual fields using the function $[\![\_]\!]_T$. The serialization of an object is done 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

[40] The in-memory representation is intentionally left unspecified. The expected behavior will be the result of a straightforward implementation.

compound types provide a function $size : \mathcal{T} \rightarrow \mathcal{I}$, which returns the number of elements stored in a given field. Let f be a non-constant[41] non-`auto` field of type t, then $[\![f]\!]_t$ is defined as[42]

- $\forall t \in \mathcal{U} \cup \{\textbf{string}\}.[\![f]\!]_t = \begin{cases} \texttt{0x00}, & f = \texttt{NULL} \\ [\![index(f)]\!]_{v64} & else \end{cases}$
- $[\![f]\!]_{\textbf{annotation}} = \begin{cases} \texttt{0x00 0x00}, & f = \texttt{NULL} \\ [\![baseTypeName(f)]\!]_{v64} \circ [\![index(t)]\!]_{v64} & else \end{cases}$ [43]
- $[\![\top]\!]_{\textbf{bool}} = [\![\varepsilon x.x \neq 0]\!]_{i8}$[44]
- $[\![\bot]\!]_{\textbf{bool}} = \texttt{0x00}$
- $\forall t \in \mathcal{I} \setminus \{\textbf{v64}\}.[\![f]\!]_t = f$
- $[\![f]\!]_{\textbf{v64}} = encode(f)$[45]
- $[\![f]\!]_{\textbf{f32}} = [\![f]\!]_{\textbf{f64}} = f$[46]
- $\forall g \in \mathcal{T}, n \in \mathbb{N}^+ . t = g\texttt{[}n\texttt{]} \implies [\![f]\!]_t = \bigcirc_{i=0}^{n-1} [\![f_i]\!]_g$
- $\forall g \in \mathcal{B}, n = size(f), t \in \{g\texttt{[]}, \texttt{set<}g\texttt{>}, \texttt{list<}g\texttt{>}\}.[\![f]\!]_t = [\![n]\!]_{v64} \bigcirc_{i=0}^{n-1} [\![f_i]\!]_g$
- $\forall r, s, t \in \mathcal{T}.[\![\_]\!]_{map<r,s,t>} = [\![\_]\!]_{map<r,map<s,t>>}$
- $\forall k, v \in \mathcal{T}, n = size(f), t = \texttt{map<}k, v\texttt{>}.[\![f]\!]_t = [\![n]\!]_{v64} \bigcirc_{i=0}^{n-1} [\![f.k_i]\!]_k [\![f[k_i]]\!]_v$
- $[\![rs]\!]_{RT} / [\![rs]\!]_{RF}$ = see section 5.1.

  Note that each restriction has its own serialization function. Common properties are specified and explained at the very beginning of section 5.1.
- $[\![t]\!]_{type} = \begin{cases} [\![id]\!]_{v64} \circ [\![val]\!]_t & id \in [0,4] \\ [\![id]\!]_{v64} & id \in [5,14] \\ \texttt{0x0F} \circ [\![i]\!]_{v64} \circ [\![T]\!]_{type} & t = T[i] \\ \texttt{0x11} \circ [\![T]\!]_{type} & t = T[] \\ \texttt{0x12} \circ [\![T]\!]_{type} & t = list\langle T \rangle \\ \texttt{0x13} \circ [\![T]\!]_{type} & t = set\langle T \rangle \\ \texttt{0x14} \circ [\![K]\!]_{type} \circ [\![V]\!]_{type} & t = map\langle K, V \rangle \\ [\![32 + storagePoolIndex(t)]\!]_{v64} & t \in \mathcal{U} \end{cases}$

ids of restrictions and types are listed in appendix G. The holes are intentional and enable future built-in types without breaking the file format. To do (84)

Note that the function $storagePoolIndex$ is implicitly given by the order of storage pools appearing in a file and has therefore to be computed both upon serialization and deserialization. The first $storagePoolIndex$ is 0.

[41]Constant fields are not serialized, because their value is already stored in the type declaration.

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

[43]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.

[44]The usual solution is either 0x01 or 0xFF.

[45]With encode as defined in listing 34.

[46]Assuming the float to be IEEE-754 encoded (see [iee08] §3.4), which allows for an implicit bit-wise conversion to fixed sized integer.

Note Constant length arrays take types as arguments. This is different to the specification language and enables future modifications such as constant size matrix types.

8.5 Endianness

Files are stored in network byte order, as described in RFC1700, Page 2 [RP94].

If a client is running on a little endian machine, the endianness has to be corrected, both when reading and writing files. This can be achieved by changing the implementation of $[\![\_]\!]_{i*}$- and $[\![\_]\!]_{f*}$-translations. Note that some standard libraries provide functions to read and write binary data in network byte order.

8.6 Date Example

To do (85)

This section will provide a concise example of how serialization takes place. For the sake of simplicity and brevity, we will serialize two objects of a single (simple) type into a stream. We will use the following file format:

Listing 32: Date File Format

```
Date {
  v64 date;
}
```

Further, we want to store two objects with *date* values 1 and -1. To do (86)

The first thing we have to do is to collect the objects to be stored. In a set-theory inspired notation with square brackets indicating records we would get something like:

$$Date = \{[date : 1], [date : -1]\}$$

Now we have to create storage pools. We start with the creation of type information. The date storage pool looks something like this:

$$[name : \text{"}date\text{"}, super : 0, count : 2, restr. : \emptyset, fields[$$

$$1 : [restr. : \emptyset, t : \texttt{v64}, n : \text{"}date\text{"}, [1, -1]]]$$

To do (87)

To do (88)

Now we can see, that we have a string, which has to be serialized, but is not yet in the string pool (which was empty up until now), so we create a string pool[47] as well:

$$[1 : \text{"}date\text{"}]$$

Now we can start to encode the pools: To do (89)

1. write the string block:
   $[\![[1 : \text{"}date\text{"}]]\!] = [\![1]\!]_{v64} \circ offset(\text{"}date\text{"}) \circ [\![\text{"}date\text{"}]\!]$
   $= [\![1]\!]_{v64} \circ [\![4]\!]_{i32} \circ$ `64 61 74 65`
   $=$ `01 00 00 00  04 64 61 74  65`

[47]The string pool, unlike regular storage pools is just a length encoded array of strings.

2. write the type block:

   To keep things readable, we will first encode everything but the field data:

   $\llbracket [name : "date", super : 0, count : 2, restr. : \emptyset, \cdots \rrbracket$
   $= \llbracket 1 \rrbracket_{v64} \circ \llbracket "date" \rrbracket \circ 0 \circ \llbracket 2 \rrbracket_{v64} \circ \llbracket \emptyset \rrbracket_{R[]}$
   $= 1 \circ index("date") \circ 0 \circ 2 \circ \llbracket size(\emptyset) \rrbracket_{v64}$
   $= 1 \circ 1 \circ 0 \circ 2 \circ 0 =$ `01 01 00 02 00`

   Now we can continue with the field:

   To do (90)

   $\llbracket [\cdots fields[1 : [restr. : \emptyset, t : \texttt{v64}, n : "date", [1, -1]]]] \rrbracket$
   $= \llbracket size(fields[\cdots]) \rrbracket \circ \llbracket [restr. : \emptyset, t : \texttt{v64}, n : "date", offset([1, -1])] \rrbracket \circ \llbracket [1, -1] \rrbracket$
   $= 1 \circ \llbracket \emptyset \rrbracket_{rest[]} \circ \llbracket \texttt{v64} \rrbracket_{type} \circ \llbracket "date" \rrbracket \circ offset("date.date") \circ \llbracket [1, -1] \rrbracket$
   $= 1 \circ 0 \circ \texttt{0B} \circ index("date") \circ offset("date.date") \circ \llbracket 1 \rrbracket_{v64} \circ \llbracket -1 \rrbracket_{v64}$
   $= 1 \circ 0 \circ \texttt{0B} \circ 1 \circ offset("date.date") \circ \texttt{01} \circ \texttt{FF FF FF FF FF FF FF FF FF}$
   $= 1 \circ 0 \circ \texttt{0B} \circ 1 \circ \llbracket 10 \rrbracket \circ \texttt{01} \circ \texttt{FF FF FF FF FF FF FF FF FF}$
   $= \texttt{01 00 0B 01} \circ \texttt{0A} \circ \texttt{01 FF FF FF FF FF FF FF FF FF}$
   $= \texttt{01 00 0B 01 0A 01 FF FF FF FF FF FF FF FF FF}$

   The type block is now serialized to the stream:

   ```
   01 01 00 02  00 01 00 0B  01 0A 01 FF  FF FF FF FF  FF FF FF FF
   ```

3. writing the output:

   The remaining work is just to write the string block and the type block to a file, starting with the string block, so we get:

   ```
   01 00 00 00  04 64 61 74  65 01 01 00  02 00 01 00
   0B 01 0A 01  FF FF FF FF  FF FF FF FF  FF
   ```

Deserialization of this stream is explained in section 9.2.

8.7 Map Example

In this short section will give an example on how to serialize a single map field. For the sake of brevity and simplicity, we will encode a map of type `map<i8,i8,i8>`. The maps data will be $m = (-1 \rightarrow (-2 \rightarrow -3, -3 \rightarrow -3), -2 \rightarrow (-1 \rightarrow -2))$. The entries are all negative, in order to make distinction between coded sizes and values obvious.

The coding of a map is basically a recursion on the remaining map types, consuming types from left to right. This leads to two cases:

1. The remaining map is at least binary:

   We have to store the number of keys and for each key, we have to store the key, followed by the serialization of the referred value. Remember that `map<U,V,T>` is right associative, i.e. it is treated like `map<U, map<V,T>>`.

2. There is only one type left:

   This means, that we have to encode the value of the last map and we are done.

Serialization

$[\![m]\!]$
by rule 1:
$= [\![2]\!] \circ [\![-1]\!] \circ [\![m(-1)]\!] \circ [\![-2]\!] \circ [\![m(-2)]\!]$
$=$ `02 FF` $\circ [\![m(-1)]\!] \circ [\![-2]\!] \circ [\![m(-2)]\!]$
by rule 1 on m(-1):
$=$ `02 FF` $\circ [\![2]\!] \circ [\![-2]\!] \circ [\![m(-1)(-2)]\!] \circ [\![-3]\!] \circ [\![m(-1)(-3)]\!] \circ [\![-2]\!] \circ [\![m(-2)]\!]$
$=$ `02 FF 02 FE` $\circ [\![m(-1)(-2)]\!] \circ [\![-3]\!] \circ [\![m(-1)(-3)]\!] \circ [\![-2]\!] \circ [\![m(-2)]\!]$
by rule 2 on m(-1)(-2&-3):
$=$ `02 FF 02 FE` $\circ [\![-3]\!] \circ [\![-3]\!] \circ [\![-3]\!] \circ [\![-2]\!] \circ [\![m(-2)]\!]$
$=$ `02 FF 02 FE  FD FD FD FE` $\circ [\![m(-2)]\!]$
by rule 1 on m(-2):
$=$ `02 FF 02 FE  FD FD FD FE` $\circ [\![1]\!] \circ [\![-1]\!] \circ [\![m(-2)(-1)]\!]$
$=$ `02 FF 02 FE  FD FD FD FE  01 FF` $\circ [\![m(-2)(-1)]\!]$
by rule 2:
$=$ `02 FF 02 FE  FD FD FD FE  01 FF` $\circ [\![-2]\!]$
$=$ `02 FF 02 FE  FD FD FD FE  01 FF FE`

Colored Data

(-1 ⟶ (-2⟶-3, -3⟶-3), -2 ⟶ (-1⟶-3))
`02 FF 02 FE FD FD FD FE 01 FF FE`

9 Deserialization

To do (91)

Deserialization is mostly straightforward.

The only notable property is that a binding has to ensure that no objects can be allocated that have more fields in the deserialized file than in the specification. This is important because it prevents from violating invariants of other tools in a tool chain by allocating objects through a partial view onto a type.

The general strategy is:

1. the string block is processed
2. the header of the type block is processed
3. required fields are parsed using the type and position information obtained from the respective block
4. until the end of file has been reached, goto step 1

9.1 An LL-inspired Approach to sf-Parsing

To do (92)

Various approaches to parsing have shown, that the most successful and straightforward approach to parsing a binary SKilL file is an LL(1)-style recursive descent parser. The format is designed, such that context sensitive parsing can be done easily using known parts of the state.

In this section, we will sketch a lexer-parser-implementation using notation and specification from section 8.

9.1.1 Lexer

The lexer provides the parser with tokens for

- bool
- i8, ..., i64
- v64
- f32, f64
- utf8 character arrays, as they occur in string blocks

Furthermore, the lexer shall provide an EOF token allowing the parser to stop after a type block. The EOF token can be implemented implicitly by adding a query to the lexer. For the sake of efficiency, the lexer should provide means of random access to a file.

Errors The lexer can only fail, if the parser requests a token beyond the end of file.

9.1.2 Parser State

We assume that the parser provides an implementation for the functions

- $string(i)$: The i'th string stored in the current file.
- $type(i)$: The i'th type definition stored in the current file.
- $type(n)$: The type definition of the name n, as obtained from the current file.
- $super(t)$: The super type of t or $\perp$ if none exists.
- $fields(t)$: The fields defined for type t.

We assume further, that there is are *string.next* and *type.next* functions, that add a new string or type to the state. To do (93)

Note String and type are basically arrays, while other queries should be fields of a class or record structure.

9.1.3 Parser

We use the operator "$\rightarrow$" (read as "is") to store the value of a token or rule in a variable. We use indention to indicate the scope of a loop. Parsing starts with the rule `File`.

File Files consist of string and type blocks:

```
while(!EOF)
  StringBlock
  TypeBlock
```

In order to improve error reporting, the number of already processed blocks should be stored and included in error messages.

StringBlock Using local variables `length` and `offsets` we get:

```
v64 → length
for(0 <= i <= length)
  v64 → offsets(i)

for(0 <= i <= length)
  utf8[offset(i)] → string.next
```

The string block should be handled lazily. This can be achieved by storing the offsets in the parsers state accessing the strings on demand.

TypeBlock Using the local variables `length` and `newFields` we get:

```
v64 → length
for(0 <= i <= length)
  TypeDefinition → (type.next, newFields.next)

for(f in newFields)
  FieldData
```

The `FieldData` rule is similar to the definitions in section 8.4.

TypeDefinition A heavily context-sensitive rule:)

```
string(v64) → name
if(   type(name))
  ..
else
  ..
```

The ∃type checks, whether we defined the type in a previous block. If the type has been defined in this block, an error has to be reported.

9.2 Date Example

To do (94)

Let d be the deserialization function – basically the inverse function of ⟦_⟧. We want to read the sequence we created during the serialization example in section 8.6:

```
01 00 00 00  04 64 61 74  65 01 01 00  02 00 01 00
0B 01 0A 01  FF FF FF FF  FF FF FF FF  FF
```

1. deserialzation of a string block:

   $d(010000000464617465010100\cdots)$
   →a string block starts with a v64 indicating the number of strings stored inside

   $d(01)d(0000000464617465010100\cdots)$
   →we got one string

   $d(01)d(00000004)d(64617465010100\cdots)$
   →the next string has 4 bytes, the block ends in 4 bytes

   $string[1:d(64617465)]d(0101\cdots)$
   →build the string pool with the first string block

$string[1 : "date"]d(0101 \cdots)$
$\rightarrow$we processed the string directly, because lazy evaluation makes the example rather confusing

2. deserialzation of a type block (reading the header):

   $d(01)d(0100020001000B010A01 \cdots)$
   $\rightarrow$there is one type definition in this block; read its name

   $d(01)d(00020001000B010A01 \cdots)$
   $= "date"d(00)020001000B010A01 \cdots)$
   $\rightarrow$we do not know the type "date" yet (in terms of processing the file), so we expect super type information, count, restrictions and field declarations

   $[name : "date", super : 0]d(02)d(00)d(01)d(000B010A01FF \cdots)$
   $\rightarrow$Date has no super type, thus the next field is not a local BPO and we can read until the number of fields.

   $[name : "date", super : 0, count : 2, restr. : \emptyset, fields[1 : \_]]$
   $d(00)d(0B)d(01)d(0A)01FF \cdots)$
   $\rightarrow$We did not know date yet, thus we do not know the type of the first field. Therefore the next information are the fields restrictions, type, name and offset.

   $[name : "date", super : 0, count : 2, restr. : \emptyset, fields[1 : [restr. : \emptyset, t : \texttt{v64}, n : "date", offset : 10]]]d(01FF \cdots)$
   $\rightarrow$We read all data belonging to the header of this type block, thus we know that the next 10 bytes are field data. The file ends after the next 10 Bytes, so we can continue constructing objects.

3. construction of date objects:

   $Date = \{[date : \_], [date : \_]\}d(01FF \cdots)$
   $\rightarrow$using the data from the header, we created two date objects; the only thing left is parsing field data for the date fields and setting the data accordingly

   $Date = \{[date : 1], [date : \_]\}d(FF \cdots)$
   $\rightarrow$the first field is a 1

   $Date = \{[date : 1], [date : -1]\}$
   $\rightarrow$the second field is a -1; we reached the end of the field data on the last object, so everything is fine – no additional or missing data

   Unsurprisingly, the restored objects are exactly the objects we serialized in section 8.6. To do (95)

There is a theory which states that if ever anyone discovers exactly what the Universe is for and why it is here, it will instantly disappear and be replaced by something even more bizarre and inexplicable. There is another theory which states that this has already happened.

Douglas Adams in *The Restaurant at the End of the Universe*

Part IV Future Work

Further research has to be done in the area of restrictions. Especially the effect of mixins on the usage of skill can not be predicted at this moment. To do (96)

A general purpose viewer and editor for SKilL files should be implemented.

We will look into ways of implementing type-safe unions. It has to be evaluated, whether they should be first class citizens of the SKilL type system, or if they can be represented with interfaces and hints or restrictions. We will also evaluate possible ways to support type casts, in order to allow for using language specific types, such as bit-fields.

We will introduce a general assertion restriction, which can be used to assert per instance invariants with sufficiently powerful expressions.

We will evaluate a template import statement, which can be used to import files together with a substitution. That way, one can create more complex data structures such as B-Trees. This feature is not a priority and only useful for large files and projects and requires some kind of type casts to yield the desired effect automatically.

We will further evaluate means of supporting common high level specification tasks, such as creating effective views for individual tools of a tool chain. This can either be in the form of a skill specification editor or an extended specification language.

Although there have been experiments with name spaces, the mechanism has been deferred to future versions of SKilL. This is the case, because there was no obvious easy-to-use implementation for existing language bindings. Furthermore many questions regarding name spaces remained open, such as

- Can name spaces have comments?
- Can they have hints or restrictions?
- What if multiple uses of the same name space have contradicting modifiers?
- Share name spaces a name space with type definitions?

To do (97)

Numerical limits, as described in appendix F, will be dropped as soon as the top 20 programming languages support 64bit index types.

The concept of views introduced in this version may be extended in future revisions.

Part V
Appendix

A Full Grammar of the Specification Language

⟨*file*⟩	::=	⟨*header*⟩ ⟨*definitions*⟩
⟨*header*⟩	::=	⟨*head-comment*⟩* ⟨*include*⟩*
⟨*head-comment*⟩	::=	'#' (~[\n])* '\n'
⟨*include*⟩	::=	(`include`|`with`) ⟨*string*⟩+
⟨*declaration*⟩	::=	⟨*namespace*⟩
	|	⟨*user-type*⟩
	|	⟨*enum-type*⟩
	|	⟨*interface-type*⟩
	|	⟨*field-change*⟩
⟨*namespace*⟩	::=	⟨*description*⟩ `namespace` ⟨*ID*⟩ '{' ⟨*declaration*⟩+ '}'
⟨*user-type*⟩	::=	⟨*change-modifier*⟩? ⟨*description*⟩ ⟨*ID*⟩ ((':'|`with`|`extends`) ⟨*ID*⟩)* '{' ⟨*field*⟩* '}'
⟨*change-modifier*⟩	::=	`++` | `--` | `==`
⟨*enum-type*⟩	::=	⟨*comment*⟩? `enum` ⟨*ID*⟩ '{' ⟨*ID*⟩ (',' ⟨*ID*⟩)* ';' ⟨*field*⟩* '}'
⟨*interface-type*⟩	::=	⟨*comment*⟩? `interface` ⟨*ID*⟩ ((':'|`with`|`extends`) ⟨*ID*⟩)* '{' ⟨*field*⟩* '}'
⟨*field-change*⟩	::=	⟨*change-modifier*⟩ ⟨*description*⟩ (`auto`)? ⟨*type*⟩ ⟨*ID*⟩ '.' ⟨*ID*⟩ ';'
⟨*field*⟩	::=	⟨*description*⟩ (⟨*constant*⟩|⟨*data*⟩) ';'
⟨*constant*⟩	::=	`const` ⟨*type*⟩ ⟨*ID*⟩ '=' ⟨*int*⟩
⟨*data*⟩	::=	(`auto`)? ⟨*type*⟩ ⟨*ID*⟩

Table 1: SKilL Specification

⟨*type*⟩	::=	`map` ⟨*type-multi*⟩
	|	`set` ⟨*type-single*⟩
	|	`list` ⟨*type-single*⟩
	|	⟨*array-type*⟩
⟨*type-multi*⟩	::=	’<’ ⟨*ground-type*⟩ (’,’ ⟨*ground-type*⟩)+ ’>’
⟨*type-single*⟩	::=	’<’ ⟨*ground-type*⟩ ’>’
⟨*array-type*⟩	::=	⟨*ground-type*⟩ (’[’ ⟨*INT*⟩? ’]’)?
⟨*ground-type*⟩	::=	`annotation` | ⟨*ID*⟩

Table 2: SKilL Specification Types

⟨*description*⟩	::=	⟨*comment*⟩? (⟨*restriction*⟩|⟨*hint*⟩)*
⟨*comment*⟩	::=	`/*` ⟨*comment-text*⟩ (⟨*comment-TAG*⟩ `:`? ⟨*comment-text*⟩)* `*/`
⟨*comment-text*⟩	::=	(⟨*comment-prefix*⟩ (~[`*/`|⟨*comment-TAG*⟩])*)*
⟨*comment-prefix*⟩	::=	’\n’ ⟨*whitespace*⟩ (’*’ ⟨*whitespace*⟩)?
⟨*comment-TAG*⟩	::=	’@’ [a-z]+
⟨*restriction*⟩	::=	’@’ ⟨*ID*⟩ (’(’ (⟨*r-arg*⟩ (’,’ ⟨*r-arg*⟩)*)?’)’)?
⟨*r-arg*⟩	::=	⟨*FLOAT*⟩ | ⟨*INT*⟩ | ⟨*STRING*⟩ | (⟨*ID*⟩ (’.’ ⟨*ID*⟩)*)
⟨*hint*⟩	::=	’!’ ⟨*ID*⟩

Table 3: SKilL Specification Descriptions

B Full Grammar of the Binary File Format

The notation used to specify the file format is explained in chapter 8. Additionally we use small fracture letters for blocks, to avoid collisions. This is basically a synopsis of chapters 8 and 5.1.

$$File = \mathfrak{s}_1 \circ \mathfrak{t}_1 \circ \cdots \mathfrak{s}_n \circ \mathfrak{t}_n$$
$$\mathfrak{s} = [\![n]\!]_{v64} \circ [\![end_1]\!]_{i32} \cdots [\![end_n]\!]_{i32} \circ [\![string_1]\!]_{utf8[]} \cdots [\![string_n]\!]_{utf8[]}$$
$$\mathfrak{t} = [\![n]\!]_{v64} \circ [\![t_1]\!]_{\mathbb{T}} \cdots [\![t_n]\!]_{\mathbb{T}} \circ [\![t_1.f_1]\!] \cdots [\![t_i.f_j]\!] \cdots [\![t_n.f_m]\!]$$

$$[\![t]\!]_{\mathbb{T}} = [\![name]\!]_{string} \circ [\![t]\!]_{?1} \circ [\![t]\!]_{?3} \circ [\![count]\!]_{v64} \circ [\![\#fs]\!]_{v64} \circ [\![field_1]\!]_{\mathbb{F}} \cdots [\![field_{\#fs}]\!]_{\mathbb{F}}$$
$$[\![t]\!]_{?1} = \begin{cases} [\![rs]\!]_{RT} \circ [\![super.typeID - 31]\!]_{v64} & \text{first occurrence of } t \text{ in } S \wedge \exists super \\ [\![rs]\!]_{RT} \circ [\![0]\!]_{v64} & \text{first occurrence of } t \text{ in } S \wedge \nexists super \\ \varepsilon & else \end{cases}$$

$$[\![t]\!]_{?3} = \begin{cases} [\![LBPO]\!]_{v64} & \exists super \wedge count \neq 0 \\ \varepsilon & else \end{cases}$$

$$[\![f]\!]_{\mathbb{F}} = [\![ID]\!]_{v64} \circ [\![f]\!]_{?2} \circ [\![end]\!]_{v64}$$

$$[\![t]\!]_{?2} = \begin{cases} [\![name]\!]_{string} \circ [\![rs]\!]_{RF} \circ [\![t]\!]_{TYPE} & \text{first occurrence of } f \text{ in } S \\ \varepsilon & else \end{cases}$$

To do (98)

C Variable Length Coding

Size and Length information is stored as v64, i.e. 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. Note that sign casts have to take place, because some interpretations require signed integers, e.g. fields of type v64, while others require unsigned integers, e.g. lengths and offsets in the binary encoding. The following small C++ functions will illustrate the algorithm:

Listing 33: Variable Length Decoding

```
uint64_t decode(uint8_t* v64) {
  int count = 0;
  uint64_t result = 0;
  register uint64_t bucket;
  for(; count < 8 && (*v64)&0x80; count++, v64++) {
    bucket = v64[0];
    result |= (bucket&0x7f) << (7*count);
  }
  bucket = v64[0];
  result |= (8==count?bucket:(bucket&0x7f)) << (7*count);
  return result;
}
```

Note The decode function can be improved using manual loop unrolling in most programming language compiler combinations. To do (99)

Listing 34: Variable Length Encoding

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

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

Note The encode function can be heavily simplified leading to performance improvements in all implemented programming languages. Furthermore, tests showed that precalculation of the size of results for bulk v64 data that appears, e.g. in references, especially in combination with memory mapped files leads to a further significant increase in performance.

D Error Reporting

To do (100)
To do (101)

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.

To do (102)
To do (103)

Serialization

- TBD

Deserialization

- Type names have to be checked for illegal characters. This is very important in order to detect future format extensions. Such a file has to be rejected.
- If EOF is encountered unexpectedly, an error must be reported before producing any observable result. To do (104)
- If an index into a pool is invalid[48], an error must be reported.
- If the deserialization of the data chunk of a field does not consume exactly the `sizeBytes` specified in its header, an error must be reported. This is a strong indicator for a format change.

[48]because it is larger then the last index/size of the pool

- If the serialized type information contains cycles, an error has to 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 pool contains instances which, based on their location[49] in the base pool, should be subtypes of some kind, but have no respective subtype 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. This is a strong indicator for either file corruption or a bug in the previously used back-end.
- All known constant fields of a type have to be checked before producing any observable objects of the respective type. 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 type block index, the total number of type blocks, the expected value and the actual value.
- If a serialized value violates a restriction or the invariant of a type,[50] an error must be reported as soon as this fact can be observed[51].
- If a restriction can not be parsed, because it has an odd ID but is not known to the generator, an error has to be reported that contains at least the involved type or field, location and ID. There shall be an option for the generator, which causes the generated code to raise this exception for even IDs as well.

E Levels of Language Support

The implementation of a fully tested SKilL binding is a time consuming and error prone process. In order to keep things simple, the language has been divided into three parts. The basic level can be implemented in a short period of time and suffices for many SKilL related tasks. The core level contains all common SKilL functionality. Core level bindings can usually be created by extending a basic level binding. The full level contains all SKilL features. Creating and testing an efficient full featured binding is expected to be time consuming and not relevant for many use cases. Experience shows that many designs for core level bindings can not be extended to full featured SKilL bindings. The Architecture of a full SKilL binding is explained in chapter 6. To do (105)

As a consequence, any binding may be called SKilL binding that implements the core language features. The list below shall also serve as a means of talking about the abilities provided by a specific language binding (generator).

- Basic
  - Integer types `i8` to `i64` and `v64`
  - `string` and `bool`
  - reading of files exactly matching the specification
  - writing of files exactly matching the specification
  - means of SKilL state management

[49]I.e. after a subtype but not inside another subtype
[50]Including sets containing multiple similar objects.
[51]This may in fact be never, if the field has an onDemand hint and is not used.

  - User types with sub-typing

- Core
  - `annotation`
  - Floats
  - Compound types
  - `const` and `auto` fields
  - Reading, Writing and Appending of files compatible with the specification.
  - Deferred Deserialization
  - Restrictions: Singleton, Default, Nullable
  - Hints: Lazy, Owner, ReadOnly, Ignore
  - Error reporting

- Full
  - Interfaces, if target language supports them
  - Enumerations, if target language supports them
  - Typedefs, if target language supports them
  - Views
  - other Restrictions
  - other Hints
  - Support for files exceeding the guaranteed numerical limit range[52].
  - Safe behavior of interleaving states.
  - Language dependent treatment of comments, e.g. integration into doxygen or javadoc.
  - Name mangling to allow for usage of language keywords or illegal characters (unicode) in specification files, without making a language binding impossible.
  - A public reflective interface allowing for fully generic file manipulation

F 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 can not deal with arrays with a size larger then about 2^{31}. 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 because of numerical limits of the target platform. Although, strings can have at most 2^{32} bytes of data, strings are usually represented as array of characters, therefore their length is expected to not exceed 2^{30} characters as well.

[52]depending on the target platform, this might not be desirable or might even be a for-free feature

G Numerical Constants

This section will list the translation of type IDs (as required in section 8.4) and restriction IDs (see section 5.1 and 8.4). Restrictions with even IDs can be ignored if unknown.

Type Name	Value	Hex Value
const i8	0	0x0
const i16	1	0x1
const i32	2	0x2
const i64	3	0x3
const v64	4	0x4
annotation	5	0x5
bool	6	0x6
i8	7	0x7
i16	8	0x8
i32	9	0x9
i64	10	0xA
v64	11	0xB
f32	12	0xC
f64	13	0xD
string	14	0xE
T[i]	15	0xF
T[]	17	0x11
list<T>	18	0x12
set<T>	19	0x13
map<T_1, ..., T_n>	20	0x14
T	32 + idx_T	0x20 + idx_T

Type IDs

Type Restriction	ID	Serialization
unique	0	ε
singleton	1	ε
monotone	2	ε
abstract	3	ε
default	5	$[\![default]\!]_{\mathcal{T}}$

Restriction IDs for Types

Restriction IDs for Fields

Field Restriction	ID	Serialization
nonnull	0	ε
default	1	$[\![default]\!]_{\alpha}$
range	3	$[\![min]\!]_{\alpha}[\![max]\!]_{\alpha}$
coding	5	$[\![name]\!]_{string}$
constantLengthPointer	7	$[\![name]\!]_{string}$
oneOf	9	$[\![types]\!]_{\mathcal{T}[]}$

Glossary

base type The root of a type tree, i.e. the farthest type reachable over the super type relation. 49

built-in type Any predefined type, that is not a compound type, i.e. annotations, booleans, integers, floats and strings. 20, 22

subtype If a user type A extends a type B, A is called the sub type (of B). 17, 41, 50, 64

supertype If a user type A extends a type B, B is called the super type (of A). 17, 41, 46, 50, 51

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. 17

user type Any type, that is defined by the user using a type declaration. 17, 20, 22

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. 67

Acronyms

API Application Programming Interface. 19, 38, 39

BPO Base Pool Offset. 46, 50

EXI Efficient XML Interchange Format. 7

LBPO Local Base Pool Offset. 46, 51

SKilL Serialization Killer Language. 5–10, 15, 16, 19, 20, 23, 26, 28, 32, 43, 47, 48, 55, 59–61, 64

v64 Variable length 64-bit signed integer. 20, 37, 45, 50, 62

XML Extensible Markup Language. 5–8

XSD XML Schema Definition Language. 7, 8

References

[Apa13] Apache Software Foundation. *Thrift Types*. http://thrift.apache.org/docs/types/, 2013.

[Blo08] Joshua Bloch. *Effective Java (2Nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008.

[ERW08] Jürgen Ebert, Volker Riediger, and Andreas Winter. Graph technology in reverse engineering, the tgraph approach. In Rainer Gimnich, Uwe Kaiser, Jochen Quante, and Andreas Winter, editors, *10th Workshop Software Reengineering (WSR 2008)*, volume 126 of *GI Lecture Notes in Informatics*, pages 67–81, Bonn, 2008. GI.

[Goo13] Google. *Protocol Buffers Language Guide*. https://developers.google.com/protocol-buffers/docs/proto, 2013.

[GSMT+08] Shudi Gao, C. Michael Sperberg-McQueen, Henry S. Thompson, Noah Mendelsohn, David Beech, and Murray Maloney. W3c xml schema definition language (xsd) 1.1 part 1: Structures. World Wide Web Consortium, Working Draft WD-xmlschema11-1-20080620, June 2008.

[iee08] IEEE Standard for Floating-Point Arithmetic. Technical report, Microprocessor Standards Committee of the IEEE Computer Society, 3 Park Avenue, New York, NY 10016-5997, USA, August 2008.

[ISO11] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, December 2011.

[jav13] *Javadoc Technology*. http://docs.oracle.com/javase/7/docs/technotes/guides/javadoc/index.html, 2013.

[LA13] Chris Lattner and Vikram Adve. *LLVM Language Reference Manual*. http://llvm.cs.uiuc.edu/docs/LangRef.html, 2013.

[Lam87] David Alex Lamb. Idl: Sharing intermediate representations. *ACM Trans. Program. Lang. Syst.*, 9(3):297–318, 1987.

[LYBB13] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification: Java Se, 7 Ed.* Always learning. Prentice Hall PTR, 2013.

[PGM+08] David Peterson, Shudi Gao, Ashok Malhotra, C. Michael Sperberg-McQueen, and Henry S. Thompson. W3c xml schema definition language (xsd) 1.1 part 2: Datatypes. World Wide Web Consortium, Working Draft WD-xmlschema11-2-20080620, June 2008.

[RP94] J. Reynolds and J. Postel. Assigned Numbers. RFC 1700 (Historic), October 1994. Obsoleted by RFC 3232.

[RVP06] Aoun Raza, Gunther Vogel, and Erhard Plödereder. Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering. In *Reliable Software Technologies – Ada-Europe 2006*, volume 4006 of *Lecture Notes in Computer Science*, pages 71–82. Springer Berlin Heidelberg, 2006.

[SK11] John Schneider and Takuki Kamiya. Efficient xml interchange (exi) format 1.0. Technical report, W3C - World Wide Web Consortium, http://www.w3.org/TR/2011/REC-exi-20110310/, March 2011.

[TDB+06] S. Tucker Taft, Robert A. Duff, Randall Brukardt, Erhard Plödereder, and Pascal Leroy. *Ada 2005 Reference Manual. Language and Standard Libraries - International Standard ISO/IEC 8652/1995 (E) with Technical Corrigendum 1 and Amendment 1*, volume 4348 of *Lecture Notes in Computer Science*. Springer, 2006.

[TM13] Martin Thompson and Todd Montgomery. *SBE*. http://weareadaptive.com/blog/2013/12/10/sbe-1/, 2013.

[Var14] Kenton Varda. *CapNProto*. http://kentonv.github.io/capnproto/index.html, 2014.

[vH13] Dimitri van Heesch. *Doxygen User Manual*. http://www.stack.nl/~dimitri/doxygen/manual/, 2013.

[WKR02] Andreas Winter, Bernt Kullbach, and Volker Riediger. An overview of the gxl graph exchange language. In *Revised Lectures on Software Visualization, International Seminar*, pages 324–336, London, UK, UK, 2002. Springer-Verlag.

[xml06] Extensible markup language (xml) 1.1 (second edition). W3c recommendation, W3C - World Wide Web Consortium, http://www.w3.org/TR/2006/REC-xml11-20060816/, September 2006.

To do...

☐ 1 (p. 5): Die Spezifikationssprache sollte Grundtypen, wie bisher verwenden; in der serialisierung sollte es aber nicht so sein; eventuell will man später sowas wie "Matrix f32[4][4] data; " erlauben.

☐ 2 (p. 5): Irgendwo sollte man unterbringen, dass der usability gedanke dazu führt, dass die spezifikationssprache und das binärformat eher losgelöst voneinander existieren

☐ 3 (p. 5): über die vier teile von skill reden: spec, api, internal state, sf $\longrightarrow\exists$foto

☐ 4 (p. 5): sicherstellen, dass lazy und deferred jetzt immer on demand genannt wird

☐ 5 (p. 5): falls sinnvoll zu jedem beispiel eine referenzdatei erzeugen

☐ 6 (p. 5): every example should have a binary skill file in the test suite and the name should be included in the example

☐ 7 (p. 9): cite,cite,cite

☐ 8 (p. 9): cite,cite,cite

☐ 9 (p. 10): ensure correctness of last sentence!

☐ 10 (p. 10): (ensure and state that) all examples in this publication are legal skill words; do this by creating test for every one of them

☐ 11 (p. 11): maybe map should be map<groundtype, maptype|groundtype>

☐ 12 (p. 11): das frontend soll eine option zur verfügung stellen, die see annotationen in kommentare einfügt, die auf die spezifikationsdatei verweisen, damit man die spezifikation bei größeren systemen wieder findet

☐ 13 (p. 15): semantik von namensräumen aufschreiben

☐ 14 (p. 18): auto fields müssen bei @mixin nicht vorhanden sein, da hier kein code generiert wird und der wert von auto feldern für die serialisierung und den state vollkommen gleichgültig ist; hier muss man darauf achten, dass auto-erreichbare instanzen nicht über closures in den state gelangen, sonst passieren merkwürdige dinge

☐ 15 (p. 18): Note: There can at most be one file per type containing an unmodified or "==" modified type declaration of that name. Note: "++" and "–" must not contradict each other. Note(style!): There **should** at most be one file containing change modifiers for a given name space. Note: Change modifiers are processed in the front-end. Note: "==" modified declarations have to be compatible to an unmodified appearance of the same type definition.

- ☐ 16 (p. 20): compatibilitätsregel: kleinere integer werden immer akzeptiert, größere werden nur dann akzeptiert, wenn ihre werte in den erwarteten bereich passen
- ☐ 17 (p. 20): Die API muss einen Hook zur verfügung stellen, der bei missmatch oder nicht existenz von Konstanten getriggert wird; andernfalls kann man konstanten nicht nachträglich einfügen
- ☐ 18 (p. 20): If an unknown non-zero constant is encountered in a known user type, the file has to be rejected
- ☐ 19 (p. 21): cast regel: falls eine datei eine annotation enthält aber ein typ erwartet wird, so wird die datei akzeptiert, falls alle instanzen dem erwateten typ genügen; in diesem fall ist ein append unzulässig
- ☐ 20 (p. 21): hier fehlt ein beispiel und eine begründung wofür annotations gut sind
- ☐ 21 (p. 21): durch die dazugekommene cast regel kann man annotations in der migrationsphase nutzen um sich mit manchen typen erst auseinanderzusetzen, wenn sie tatsächlich gebraucht werden
- ☐ 22 (p. 21): hier muss man noch mal darauf hinweisen, dass sich formate in einem ständigen fluss befinden, mit dem man umgehen muss
- ☐ 23 (p. 21): annotations dürfen beliebige typen ersetzen, solange diese kompatibel sind; das führt bei containern o.ä. dazu, dass das feld niemals einen nicht-default wert enthalten darf, weil sonst der cast fehlschlägt. ⇒annotations sind quasi das top element des typhalbverbands
- ☐ 24 (p. 21): having implicit cast from annotation to actual type is an important feature because it is required when it comes to people learning that interfaces can have regular types as base types
- ☐ 25 (p. 22): this is no longer the case; strings must be unified if they reside in memory at the same time, but this leaves potential copies of user strings; this is a bit more relaxed then unique, but it is necessary, because otherwise all strings had to be deserialized eventually.
- ☐ 26 (p. 22): explain(may be moved to other document): the problem with having containers as base types is that they would not have a reference semantics but a value semantics, which might be very confusing especially for java programmers; the old way is the better solution; wo should however keep the n-nary maps a concept of the specification language and force the binary files and the api to treat them as right associative binary maps

☐ 27 (p. 22): note: having compound type arguments is a bit like free variables in logic, because users might think that T[] for any T is a Type, like T itself, but that's not the way skill treats types

☐ 28 (p. 22): make clear that containers are stored in a flat way into an object. This is suprisingly what most users require and ther HAS to be a way to do this in the current encoding schema. If one wanted to share such objects, they can me moved into their own type declaration.

☐ 29 (p. 22): map a b c = map a map b c

☐ 30 (p. 23): interfaces; interfaces can either be used to group properties of subtypes or to describe properties of types; both can not be mixed, because the mapping onto regular types would no longer be monotone (the intersection of two unrelated supertypes is annotation, but super types would be "deeper"; thats a problem)

☐ 31 (p. 23): interfaces as subtypes of regular types are a promise, that all implementations are subtypes of the regular type as well; therefore only one regular type can be inherited

☐ 32 (p. 23): nach der erweiterung müssten hier deutlich mehr regeln stehen

☐ 33 (p. 23): ad interfaces: in der IR geplättete interfaces sind ein usability Problem, weil man aller Wahrscheinlichkeit nach ungeprüfte typecasts im binding als user braucht, weil man keinen typ hat, der die existenz eines feldes garantiert; das sollte man eher vermeiden

☐ 34 (p. 23): interfaces dürfen nur kommentare haben, aber weder hints noch restrictions

☐ 35 (p. 23): ad enums: wenn man defaults hat sind enums nurnoch darstellungssache; im sf sind sie so teuer wie normale integer, weil sie als benannte subtypen per reference quasi über ihren ordinalen wert serialisert werden, was der C lösung entspricht, nur dass man die usability von Java enums bekommt:)

☐ 36 (p. 24): move and distribute

☐ 37 (p. 24): move and distribute

☐ 38 (p. 26): erklären, was genau typsicher für uns bedeutet; eventuell auch etwas über zustandsübergänge sagen, weil das auch etwas anders ist als in üblichen sprachdefinitionen

☐ 39 (p. 26): es gibt kein write! es gibt nur state management funktionen, wie compress, die auf einer leeren datei das selbe sind, wie früher write

☐ 40 (p. 26): there shall be a both, a pragma restriction and a pragma hint; both intended for exploratory purpose of the skill format; a warning shall be emitted upon use of the front-end if a pragma is encountered to prohibit use in production code

- ☐ 41 (p. 30): translation required!
- ☐ 42 (p. 30): (hier muss man die skill-spec anpassen)
- ☐ 43 (p. 30): hier sollte eine liste von default werten für den fall stehen, dass keine default restriction angegeben ist; die antwort ist im prinzip immer 0, es sei denn es gibt kein 0
- ☐ 44 (p. 30): consider: The section about default values somehow disappeared from the TR. Default values need to be specified again.

To do (??)

In contrast to the old definition, which was basically equivalent to the java byte code default values definition, compound types shall have empty collecctions as default instances. This has no impact on the ABI, because serialization of empty collections and null pointers is equivalent.

- ☐ 46 (p. 32): doch, bitvector!
- ☐ 47 (p. 32): hier (oder vermutlich in einem anhang) sollte man als Beispiel "bitvector" verwenden, um sets<T> oder bool[] zu kodieren
- ☐ 48 (p. 36): Das hier ist in wirklichkeit ein coding "i64"
- ☐ 49 (p. 36): mixins erklären; das sollte den platform mechanismus abbilden; man muss aber platformspezifische escapes definieren;hier sollte man vielleicht eine *.language.mixin format als seiten-spezifikation definieren um ausreichend customization zu bieten; anm.: mixins sind ein sehr fortgeschrittenes feature, sowohl für binding, aber vor allem für den user
- ☐ 50 (p. 36): auch deserialization? eigentlich sollte unifikation durch den hint vorgenommen werden und von der restriction erhalten
- ☐ 51 (p. 39): no: its a list of arrays-> selber memory footprint, aber (konstant) mehr indirektionen
- ☐ 52 (p. 39): this chapter requires a rewrite
- ☐ 53 (p. 39): section state transitions: create, read, write, append; hier sollte man einen graphen malen, welche übergänge zulässig sind; außerdem sollte man erwähnen, dass weder write noch append daten wegwerfen dürfen
- ☐ 54 (p. 39): state that operations like write can not be performed in parallel to state modifications in general
- ☐ 55 (p. 39): reorganize this whole section; it should explain exactly how skill states should work, how state transitions are realized and how implementation can be done in parallel(on SMP nodes)

☐ 56 (p. 39): falls das noch nicht so ist, dann muss man hier fordern, dass die datenstrukturen in einer inlinefreundlichen weise organisiert werden, aber nicht direkt exportiert, damit man bei änderungen, wie z.B. !distributed nicht den anwendungscode ändern muss, sondern die arbeit vom compiler für einen übernommen wird

☐ 57 (p. 39): es gibt im prinzip zwei ansätze: den kernel/typecontext (generischer) und den generated; beide haben vor und nachteile; der kernel ist nur möglich, wenn es inlining zur linktime gibt, sonst macht er performance bedingt keinen sinn

☐ 58 (p. 39): SKilL IDs dürfen nur für serialisierte daten vergeben werden; das sorgt dafür, dass der umgang einheitlich bleibt

☐ 59 (p. 39): Pools sollten auf ihren state verweisen, damit man auch mit einem einzelnen pool in der lage ist den kompletten state zu speichern; scheint auch nutzbarer zu sein

☐ 60 (p. 39): states sollen drei funktionen zur verfügung stellen: check, closure und flush; check testet restrictions, closure fügt alle transitiv erreichbaren instanzen dem state hinzu und flush macht check und closure, löscht alle als gelöscht markierten instanzen und schreibt die änderungen auf die platte

☐ 61 (p. 39): es muss eine state.add und eine state.delete methode geben; jede instanz sollte zudem eine delete methode haben und jeder typ eine add methode, die aber an denn state weiterleitet (andernfalls ist der typ nicht notwendigerweise korrekt abgebildet

☐ 62 (p. 39): die kernel sache funktioniert nur, wenn man auch in der lage ist die speicherrepräsentation zu kontrollieren und damit auch zu inlinen; das könnte in java trotzdem funktionieren, wenn die deoptimization zuverlässig genug arbeitet oder trotz jar nur eine anbindung existiert; dann wäre es als lösung für den JIT overhead aber grob gelogen

☐ 63 (p. 39): das lese/schreibe API sollte sich an C-Style files orientieren; man sollte den mode beim öffnen wählen, es sollte außerdem eine flush und eine close methode geben; außerdem gibts hier ein paar weitere, wie z.B. ein explizites check

☐ 64 (p. 39): ein State sollte sich wie ein File und wie eine Collection aus Typen verhalten

☐ 65 (p. 39): Typen sollten sich wie eine Collection aus Instanzen und eine Collection aus Feldern verhalten

☐ 66 (p. 40): Felder verhalten sich wie eine Collection aus den Feldwerten der entsprechenden Instanzen

☐ 67 (p. 40): hier sollte man etwas über die "hairy type hierarchy" sagen: es hat jeder bekannte typ einen untertyp, der alle unbekannten untertypen repräsentiert

☐ 68 (p. 40): hier sollte man was über 64bit unsigned integer indices sagen

☐ 69 (p. 42): in wirklichkeit sind storage pools natürlich keine (resizable) arrays, sondern resizeble arrays aus festen arrays, damit man sich beim resizen die kopie spart; irgendwas in der richtung sollte man machen. es ist nämlich nicht möglich, dass man ein objekt verschieben muss; außerdem sollte man löschen sowieso erst zum Zeitpunkt der serialisierung festklopfen, oder wenn der user flush() sagt

☐ 70 (p. 44): state somewhere, that although skill is casing agnostic, casing will influence the generated binding; this is especially true, because the front-end will infer word boundaries based on casing and the generator will use these boundaries to create language specific identifiers

☐ 71 (p. 44): als allererstes sollte direkt vor diesem kapitel erklärt werden, wass ein abstrakter state und was abstrakte objekte sind, damit man hier eine notation hat um serialisierung zu erklären; da sollte auch sowas drin stehen wie σ ist die menge aller U und u die menge aller felder

☐ 72 (p. 44): talk about state transition system with transitions: c,r,m,a,w,order

☐ 73 (p. 44): die regel mit aufnehmen, dass felder, die nur default werte haben nicht serialisiert werden, wenn sie zuvor nicht existierten. das ist wichtig um z.b. in IML tote datenformatsteile los zu werden

☐ 74 (p. 44): strings dürfen zusammengelegt werden, müssen aber nicht; das verhindert, dass man beim schreiben strings aus der datei lesen muss

☐ 75 (p. 44): enums brauchen in der serialisierung präfixe, sonst kollidieren sie eventuell mit anderen typen; ⇒enum namen wie namespaces behandeln!

☐ 76 (p. 44): specify that serialization has to be able to deal with (skill) type names containing '.' characters, in order to allow future revisions to implement name spaces without having to break old implementations

☐ 77 (p. 47): das wort (L)BPSI sollte aus der arbeit verbannt werden; man sollte von offsets reden und die offsets sollten so gewählt sein, dass er für den basepool implizit 0 ist und keiner verschiebung bedarf; man sollte ein bild malen, dass die verschiebung der subPools zeigt

- ☐ 78 (p. 47): replace last sentence by something obtained from IML.sf
- ☐ 79 (p. 48): ensure that this chapter is no plain contradiction to statements above
- ☐ 80 (p. 50): grammar of examples is wrong! provide links to the testsuite as well
- ☐ 81 (p. 50): prüfen! das sollte sowieso BPO lauten
- ☐ 82 (p. 51): write something about sub-storage pools
- ☐ 83 (p. 51): make sure this is consistent with the actual file format
- ☐ 84 (p. 52): hier ganz besonders aufpassen!
- ☐ 85 (p. 53): wenn die typen anders serialisiert werden, dann braucht man insbesondere bei maps einen test; das führt dazu, dass alte dateien mit neuen nicht mehr kompatibel sind!
- ☐ 86 (p. 53): umbennenen! z.b. ”age”
- ☐ 87 (p. 53): call it age; use @min(-1) and document that by saying -1 is the age of an unborn child
- ☐ 88 (p. 53): phase 1: collect instances
- ☐ 89 (p. 53): phase 2 : instances are, already in pools, but age restriction has to be checked!
- ☐ 90 (p. 54): this can be heavily simplified by calling the current stream s.
- ☐ 91 (p. 55): this is no longer correct!!!
- ☐ 92 (p. 55): das ganze kapitel muss man neu machen; viele von den aussagen stimmen so nicht, weil die deserialisierung etwas komplizierter wird
- ☐ 93 (p. 56): das LL-Parsing Kapitel sollte man streichen; statt dessen sollte man ein Kapitel mit Implementierungstipps basteln, das gute Ideeen aus veröffentlichungen zusammensammelt und zusammenfasst.
- ☐ 94 (p. 57): hier könnte man eventuell flussdiagramme verwenden; das ist leichter zu verstehen und man braucht nicht den formalismus des states
- ☐ 95 (p. 58): das hier ist jetzt genauso age; wenn man vorne was ändert muss man das hier berücksichtigen, insbesondere wenn man eine restriction verwendet
- ☐ 96 (p. 59): lazy age example

- ☐ 97 (p. 59): mehr praktische nutzung muss abgewartet werden
- ☐ 98 (p. 62): remove specification of namespaces; note: keep selector in default, because enums create mini namespaces; enums verwenden den selben encoding mechanismus wie namespaces für instanznamen
- ☐ 99 (p. 62): copy grammar for fields and restrictions
- ☐ 100 (p. 63): this is not implemented ATM! provide implementation, create benchmark and update this specification
- ☐ 101 (p. 63): diese datei sollte man auf teilen
- ☐ 102 (p. 63): check error reporting to reflect the changes in skills behavior
- ☐ 103 (p. 63): note that the earliest point to check restrictions is at the end of a type block; suggest that applicability of restrictions shall be checked after each block (to indicate the tool violating the restriction) and semantic checks shall be done as last action of the read operation in order to allow for as much parallelism as possible
- ☐ 104 (p. 63): constant have to be checked as soon as they are encountered
- ☐ 105 (p. 64): This is less restrictive then claiming that no offset may point after the EOF. This way partially lost data can still be processed.
- ☐ 106 (p. ??): cite students:)