The SKilL Language V1.0

Timm Felden

TR 2017/01

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.

This is an updated version of SKilL TR13[Fel13].

Acknowledgements

Main critics: Erhard Plödereder and Martin Wittiger.
Additional critics: Dominik Bruhn[1], Dennis Przytarski[2], Martin Kaistra[3].

Contents

I Specification Language **3**

1 Introduction **3**
1.1 Scientific Contributions 4
1.2 Outline 4
1.3 Related Work 5
1.4 Notation 7

2 Syntax **8**
2.1 Reserved Words 8
2.2 The Grammar 8
2.3 Examples 10
2.4 Style Guide 12

3 Semantics **13**
3.1 A Specification File 13
3.2 Includes 14
3.3 Type Declarations 14
3.4 Field Declarations 15
3.5 Field Types 16
3.6 Name Resolution 17

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

4 The Type System **17**
4.1 Built-In Types . . . 18
4.2 Compound Types . . . 20
4.3 User Types . . . 20
4.4 Fancy Types . . . 21
4.5 Default Values . . . 24
4.6 Examples . . . 25

5 Type Annotations **25**
5.1 Restrictions . . . 25
5.2 Hints . . . 32

II Binary File Format **38**

6 Impact of changing Specifications **38**

7 Serialization **39**
7.1 Steps of the Serialization Process . . . 39
7.2 General File Layout . . . 39
7.3 Storage Pools . . . 44
7.4 Serialization of Field Data . . . 47
7.5 Endianness . . . 48
7.6 Age Example . . . 48
7.7 Map Example . . . 50

8 Deserialization **50**
8.1 Age Example . . . 51

III Future Work **53**

IV Appendix **54**

A Full Grammar of the Specification Language **54**

B Full Grammar of the Binary File Format **56**

C Variable-Length Coding **56**

D Error Reporting **58**

E Levels of Language Support **59**

F Numerical Limits **60**

G Numerical Constants **61**

Glossary **62**

Acronyms **62**

Panta rhei (everything flows)

Heraclitus of Ephesus

Part I
Specification Language

1 Introduction

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 These problems often cause software engineers to stick with outdated tools or even programming languages, thus causing a lot of frustration.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 come to an end.

The approach presented in this paper provides a means of platform and language-independent specification of serializable data structures and, therefore, a safe way of interaction between old and new tools in a toolchain, 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 toolchain language-independent because we believe that the best language a programmer can use to write a new tool is the language that he likes best. 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.

Firstly, this goal is achieved by 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.

Furthermore, a formalized mapping of specified types to a bitwise representation of stored objects. The mapping is very compact and therefore scalable, of simple structure and therefore easily bound to a new language. The type system is serialized to 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[4].

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 Serialization Killer Language (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 allows files containing data of arbitrary types to be processed. If the data stored in the file is not used by a tool, it does not have to pay for it with execution time or memory. Furthermore, a tool does not have to know the whole intermediate representation of a toolchain but only the parts it is going to use in order to achieve its goals. The expected file sizes range from one megabyte to several gigabytes while having virtually no relevant numerical limits in the file format[5]. Please note that the SKilL file format is a lot more compact than 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 contained three pointers on average, the file size

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

[5]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.

would still be around one 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.

1.1 Scientific Contributions

This section is a very concise representation of contributions. The suggested serialization format and serialization language offer all the following features in a single product:

- a small footprint and, therefore, high decoding speeds
- a fully reflective type encoding
- type-safe[6] storage of references both to known and unknown types[7]
- a rich type system providing, among others, references, containers, single inheritance and extension points
- the specification language is modular[8] 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 SKilL to interact with it. Also, it allows programmers to write tools in the language they know best[9]
- 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

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

1.2 Outline

The specification has been split into three parts. Part I describes 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 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 contains an outlook on future developments as well as rather technical lists and tables that mainly provide a concise overview and clarification.

[6] In the sense that one cannot get a number if one expected an array.

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

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

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

1.3 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[10].
+ It is easier to write a libXML for a new language than to write a SKilL back-end[11].
- 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 a tree, whereas a SKilL file contains any number of unrestricted graphs.
- 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 cannot be appended with new data.

On XML efficiency

There are some very efficient XML implementations, e.g. `http://tibleiz.net/asm-xml/index.html`. Interestingly, they warn you "Remember: if you really need speed, do not use XML." (as of 21.7.16). Their speed is comparable to current SKilL implementations in terms of MB/sec. However, the interesting rate is Objects/sec. In this category, SKilL outperforms XML, in the toolchain intermediate representation scenario due to smaller file sizes. Furthermore, XML can never be parsed partially.

XML Schema definitions

The SKilL 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 taking a look which one better fits one's needs.

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

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 similarly to the API used by SKilL implementations. In case of Java, the mechanism is even part of the standard library. The downside is that, to the best of our knowledge, this has been implemented only 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 they appear in the process of analysing medium size programs. The TGraphs library[ERW08] claims to use GXL for communication with other toolchains. Note that the GXL Graph Model could be expressed as a SKilL specification, replacing XML with binary SKilL as data representation.

YAML

Shares some characteristics with XML, except that it is significantly less widespread and seems to have died years ago[BKEdN][SSV13].

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 than it was at the time of its creation. Tools like memory mapped files or IEEE-754 floats simply did not exist at that time.

Apache Thrift, Protocol Buffers and others

Thrift states that there is no sub-typing (see [Apa13][12]). Protocol Buffers (see [Goo13]) do not seem 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).

[12] 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[13] 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 or 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 handcrafted formats, they serve as a guiding example in many ways.

Language-Specific Serialization

Language-specific serialization is language specific and, therefore, cannot 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.

Interestingly, items 74 and 75 in Effective Java 2[Blo08] warn you not to use native serialization recommending the use of handwritten serialization mechanism where possible instead. 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 the previous version of SKilL[Fel13]. Implementations can be found at `https://github.com/skill-lang/skill`. Descriptions of implementations can be found in [Prz14, Rot15, Ung14, Har14].

1.4 Notation

Code and references to code will be written in a `typewriter font`. Full pieces of code are grouped into listings. All examples are part of the SKilL test suite. The heading of the block is the respective file name. Most files are used as front-end tests.

Listing 1: "Example Listing"

```
Some Code {
  ... <- anything 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. $\mathcal{T}$.

[13] 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 7.

We use the tokens `<id>`, `<string>`, `<int>`, `<float>` and `<comment>`. They equal C11-style[14] 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[15]. Usage of `$` and `_` characters is generally discouraged. We use a comment token because we want to emit the comments in the generated code in order to integrate the generated code 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 Scala, 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.

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

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

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

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

FIELD :=
  DESCRIPTION
```

[14] [ISO11] Annex D

[15] 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).[16]

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

2.3 Examples

Listing 2: Running Example

```
/** 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 */
  Location 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 and Includes

```
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, than just plain values or references. In the current version, there is support for sets, maps, lists and arrays:

Listing 7: Container Example

```
/** 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

```
÷ {
  ÷ †;
  ÷ €;
}
```

2.4 Style Guide

This section provides some general hints on how to write readable specifications. A uniform appearance of specifications will improve 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 be 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 it 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 it 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 indentation level.

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

Separate field declarations with comments, restrictions or hints by one blank line from other field declarations This rule turned out to be very useful. If neither comments, hints nor restrictions are used, the specification may choose to skip blank lines. Not using comments is discouraged. Separating subsequent type definitions by 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 subdivided into several files consisting of small logical units are more accessible using just a text editor than a single large file. Splitting up a specification enables tool builders to omit parts that are 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 in turn 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 declaration's 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 sections 5.1 and 5.2.

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 through `include`-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 declaration consists of at least a name and a body containing field declarations.

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[17] to serve as a natural-language description of the respective entity. This approach enables users to get tool-tips in an IDE showing them this documentation. Comments will ignore preceding whitespace and '*' tokens. They end with '*/' and they cannot be nested. Comments start with a text and can be followed by tags. Valid tags are

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

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

- @note *message*
- @todo *message*

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 behaviour in the presence of other tags is unspecified. Ill-formed tags must not cause a binding generator to fail. Casing of tags is ignored. A double colon following a tag is discarded. If a valid type or field is used as an 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] and javadoc [jav13].

3.3.2 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 field to those above 23. Hints can be used to optimize the generated language binding. Restrictions and hints are covered by section 5.

3.3.3 Subtypes

A subtype of a user type can be declared by appending the keyword `with`[18] and the supertype's 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.

[18] Alternatively, `':'` or `'extends'` can be used.

3.4.1 Constants

A `const` field can be used in order to create guards or version numbers. An API should provide a hook that is triggered if a constant is encountered that was not expected, is missing or has an unexpected value. Otherwise, if an unknown non-zero constant is encountered in a known user type, the file has to be rejected. 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 tools 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, to simplify the implementation of 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 to do with the **auto** type declaration of C++.

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[19], 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 perspective of serialization, equivalent to length-encoded arrays. Their main purpose is to increase the usability of the generated Application Programming Interface (API). These containers have shown 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 whose type could not be known at the time the annotation field is specified.

3.6 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 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 correctly 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 and do not appear in the same specification.

A generated language binding has to provide means[20] 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

Both the description language and the file format are 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 before an observable result is created.

These properties require some form of language-independent type system[21], which is described briefly in this section. The general layout of the type system is visualized

[19] 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 be solved by capitalization.

[20] Usually via reflection.

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

in Fig. 1.

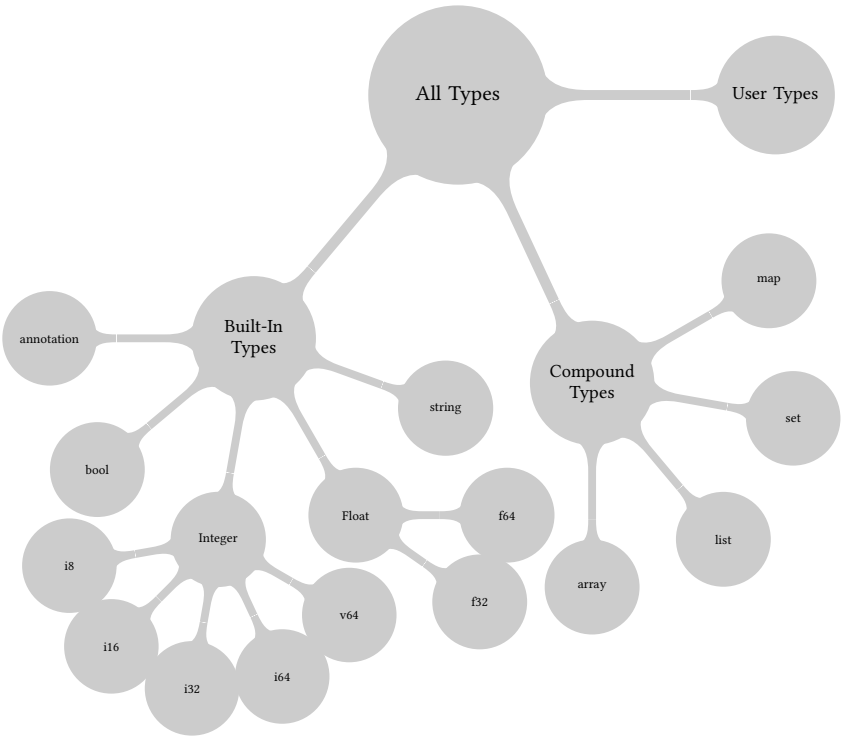

Figure 1: Layout of the Type System

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. $\{$`i8`, `i16`, `i32`, `i64`, `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 flavours, fixed length and variable length. For now, there is 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.

Conversion Implementations may accept smaller integers without checking and larger integers if all stored values can be represented in the smaller range.

Booleans

Booleans can store the values true($\top$) and false($\bot$). 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[22]. Therefore, language bindings shall provide means of inspecting whether the type of the object behind an annotation is known.

Conversion If an annotation is encountered but a certain type is expected, the annotation can be converted if all stored values are either of that type or `null`.

Strings

Strings are conceptually a variable-length sequence of utf8-encoded Unicode characters. The in-memory representation will try to make use of language features such as `java.lang.String`. The serialization is described in section 7. If a language demands NULL-termination in strings, the language binding will ensure this 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 try its best at unifying strings. However, this behaviour is only guaranteed in case of strings used as type and field names.

NULL Pointer

Fields of type strings, annotations or a user type can store a NULL pointer. These types are nullable by default 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], particularly §3.4.

[22]The reflection mechanism enables other solutions, but raising an exception is the most obvious reaction.

4.2 Compound Types

The language offers several compound types. Sets, lists and variable-length arrays[23] are 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 resizeable. 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 can be thought of as serializable partial functions. Hence, they have two or more type arguments.

Note 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. JVM-based implementations for instance use 32-bit signed integers, because this type is used for array indexing. See appendix F for more details.

Note Having compound types as flat field data in contrast to shared data with reference semantics is clearly a design decision[24]. Generalization of compound types to shared data semantics would have been a straightforward generalization. Likewise, the n-ary maps concept is available in the specification language only[25] and has been introduced to ease the modelling of data structures[26].

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: Appointment Example

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

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

  /** the name of the room. */
  string room;
}
```

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

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

[24] Surprisingly, this seems to be the most common use-case. If a shared container is required, it can simply be wrapped with a user type.

[25] We specified that $\forall a, b, c.$`map<a, b, c>` $= map(a, map(b, c))$

[26] 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

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 have to be taken into account as well.
- The subtype relation is a partial order 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 through include commands.
- The type names must be unique in the context of all[27] types.

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. Most SKilL implementations may choose to use the canonical flattening described in the sections below. If a programming language offers similar language features, they are expected to be used in the generated API.

4.4.1 Interfaces

Interfaces are intended to enable grouping of properties. In the same way that interfaces in Java provide a promise of availability of certain methods, interfaces in SKilL provide a promise of availability of certain fields.

Unlike interfaces in most modern languages, interfaces in SKilL are allowed to inherit from regular type definitions. This behaviour is possible because they do not exist in binary files. Integration of this feature into a language-specific API is undefined. Regular type definitions as well as interfaces may inherit an arbitrary amount of interfaces, as long as the type hierarchy forms a directed acyclic graph. If the same interface is inherited multiple times, it is treated as if inherited once. Inheritance of interfaces is transitive. Interface definitions can be empty.

If an interface does not inherit from a regular type definition, its super type becomes `annotation`. If an interface inherits from a regular type *T*, then all transitively

[27] 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.

inherited interfaces may inherit only regular types that are either T, a super type of T or `annotation`.

Interface definitions in the specification may have a comment but must not have hints or restrictions. The usage of hints and restrictions is ruled illegal, as there is no way to ensure that interfaces exist in any way in a generated implementation.

Projection Field declarations using interfaces in binary files are projected onto the lowest regular type inherited by the interface. Fields of interface declarations are moved to direct subtypes recursively.

4.4.2 Enums

Enumerations in SKilL start with a list of instances, followed by an optional list of field declarations. The binary representation of an enum-typed field is a small variable-length integer, i.e. usage of enums is very efficient.

Enum definitions in the specification may have a comment but must not have hints or restrictions. The usage of hints and restrictions is ruled illegal, because they may break the projection and make code generation too complicated.

Projection Enums create an abstract regular type definition containing the field definitions of the enum. This type definition has a default restriction using the first instance as the default value. Instances are projected onto singleton-restricted subtypes using the naming convention $enuminstance$.

Note that projected enum types do not have a OneOf-Restriction, because this restriction would make adding another instance to the definition very painful.

4.4.3 Typedefs

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

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 be used syntactically to nest containers inside of containers. This is forbidden for coding reasons and, therefore, will 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 argument resulting from typedef is illegal. Furthermore, typedefs cannot be used as super types. This restriction is for readability only.

Typedefs are syntactic sugar which is not exported to the binary file. The front-end shall provide a mechanism to project away all typedefs by replacing occurrences with the definition[28].

4.4.4 Views

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

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

Example:

Listing 10: View Example

```
A { A a; }
B : A {
  /* rename and retype */
  view A.a as
  B b;
  /* rename a second time */
  view a as
  A c;
}
```

Views can be used to rename and retype fields. Views will forward all access to the viewed field, thus they will inherit all its properties, except for its comment. This feature can be used in two scenarios:

Retyping is primarily useful if two companion hierarchies exist which refer to each other. Take for example 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 cannot be modified for whatever reason. Renaming may also be required if input from multiple independent tools is 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 toolchain, 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[29], while keeping the file format and, thereby, old tools.

An additional benefit of the view concept over TR13 is that renaming allows a 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 cannot change restrictions or hints of a field. This feature was dropped, as it would have required introducing views into the binary file format and, thereby, it would have increased complexity a lot. Views inherit all restrictions and hints of the viewed field except `!hide` (see §5.2.3).

[29] Using a `!hide` on the actual declaration

Views should be used with care because they may confuse people, especially if they are working with multiple specifications or language bindings. Views should not be used in enum declarations.

4.4.5 Customizations

```
'custom' <id>
(!<id> (<string>? | '(' <string>+ ')'))*
<string> <id> ';'
```

Example:

Listing 11: Language Custom Fields

```
CustomFields {
  custom ada
  !with "RFG.Node"
  "RFG.Node" node;

  custom java
  !import "my.import"
  !modifier "public synchronized"
  "Object" any;
}
```

Language custom fields are an extension on the idea of auto fields. They can be used to improve the integration of a generated API into a tool's environment.

The first *id* behind the custom keyword will be interpreted as the name of a code generator. After that, a list of options can be given using a hint-like syntax. Options use string arguments to provide the generator with flexibility. Then, a string will be used to specify the type of the field in a language-specific way. Finally, the generated field will have a name, like all other fields, that will be checked for name clashes with regular fields. Language generators may introduce further arbitrary restrictions of this feature. Treatment of custom fields is optional in the sense that generators may ignore them despite being responsible for the target language.

4.5 Default Values

Default values are:

Type	Value
integers	0
floats	0.0
enums	first definition
containers	$\emptyset$
others	null

Note that language bindings may expose other default values in their APIs. For instance, the current Java implementation uses the language default values, i.e. containers are by default null-pointers.

4.6 Examples

This section briefly explains some examples of ill-formed type declarations.

Listing 12: Legal Super Types

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

Error: The built-in type “string” cannot be subclassed.

Listing 13: Legal Type Names

```
/** The german word for "car". */
Auto {
  /* Keyword detection is case sensitive, while type
     names are not; use "Auto" instead. */
  auto previousCar;
}
```

Error: Expected field name in field declaration.

Listing 14: Usage of Unknown Types

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

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

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.

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. If Restrictions are used on compound types, they expand to the components of the respective compound type. Restrictions cannot be combined with map-typed fields. The specification of restrictions is mostly written from a user’s perspective. Nonetheless, specifying serialization of restrictions is a necessary lookahead to the next chapters.

Restrictions are serialized to assert their properties. The serialization mechanism and an optional recovery strategy are 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 $[\![ID]\!]_{v64}$[30]. Field and type restrictions do not share a common ID range and are explained in their respective subsections. 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 toolchain. Hence, fulfilling restrictions is the duty of the creator of data. In consequence, the validity of restrictions is checked prior to writes. Thus, serialized restrictions can be assumed correct. Restrictions that may be skipped by incomplete implementations necessarily have 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 does not implement a restriction, it has to warn the user about that fact at generation-time of the binding. A SKilL implementation 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.

Restrictions can be obtained both by reading a file or by adding them to the specification prior to the 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. Appending in the presence of unknown restrictions is illegal.

Furthermore, the expected API behaviour is to provide means of explicitly checking restrictions at any given time for a given state. Deferring checking to a language's type system may not be possible because it may deprive the user of the capability to create an object graph at all. For example, look at the nonnull restriction and think about how to instantiate a tree structure. Although there are solutions to the problem, they usually involve more complicated variants of the type hierarchy and the allocation style alike. Restrictions will be checked implicitly after read a file and before revealing the state to the user and, respectively, 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 cannot damage existing data.

Reading a field that does not comply with the specified or serialized restrictions renders the field partial (see section 7.2.6).

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 supertypes 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 merging after arbitrary opera-

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

tions, 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 15: Unique Example

```
@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 represented by singleton subtypes of the enum types. This enables enum instances with fields.

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

Listing 16: Singleton Example

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

Listing 17: Enum like

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

is, besides the generated API, equivalent to

Listing 18: Enum direct

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

  string name;
  i8 index;
}
```

5.1.1.3 Monotone Instances of this type cannot be deleted. Furthermore, instances cannot 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 wherever 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[31].

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

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

ID	2
Applies to	base
Serialization	ε
Recovery	add to file

Listing 19: Monotone Example

```
@Monotone PublicPost {
  string message;
}

/** this is monotone as well */
PrivatePost extends PublicPost {
  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. Consequently, it does not apply to subtypes.

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 restricted type, that is a subtype of this type[32]. The default value is inherited by sub types, but can be changed explicitly for any of them. It can be overwritten by

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

[32] Otherwise, the single instance cannot be a legal default value.

field default restrictions.

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

Listing 20: Type Default Example

```
@default(Metric)
Units {
  string length;
  string time;
}

@singleton
Metric : Units {}
```

5.1.2 Field Restrictions

Field restrictions use the type α to denote the field's type.

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	`nonnull`:
Serialization	ε
Recovery	add to file

Listing 21: Nonnull Example

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

5.1.2.2 Default This restriction can be used to change default values for all ground types[33]. Defaults for user types and annotations work in the same way as the default type restriction (see 5.1.1.5). They are serialized via their type instead of their value.

Implicit default values for types are described in section 4.5.

[33] A future revision of the language may allow a more Java style syntactic sugar for default values in the spirit of `string msg = "Hello World!";`

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

Listing 22: Default Example

```
/** A multi-graph with a weighted edges and labeled
    nodes. */
Graph {
  Node[] nodes;
  Edge[] 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. The range must not be empty.

ID	3
Applies to	Integer, Float
Signatures	`range(min, max, boundaries)`: $\alpha \times \alpha \times string^? \times string^?$ `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 23: Range Example

```
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 field's data chunks are encoded using the argument coding. No codings are specified by SKilL. Potential arguments include "bitvector", "zip" or "lzma" or some sort of encrypting container. 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. Therefore, coding is designed to allow skipping fields if the coding is unsupported.

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

Listing 24: Coding Examples

```
ToolDescription {

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

5.1.2.5 Constant-Length Pointer 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.

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

Listing 25: Constant Length Pointer Example

```
/* stored points-to information may exceed the available
   main memory, thus we access it directly from disk */
PointsToTargets {
  ...

  @constantLengthPointer
  PointsToSet targets;
}
typedef PointsToSet set<PointsToTargets>;
```

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

Listing 26: OneOf Examples

```
/* this type definition is a union of A and B */
typedef poorMansUnion
  @oneOf(A, B)
  annotation;

A{}
B{}
```

5.2 Hints

Hints are annotations that start with a single `!` and are followed by a case-insensitive hint name. Hints are used to optimize the behaviour 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. Further, language binding generators shall provide an option that adds a hint to all applicable declarations.

5.2.1 Distributed

Can be used on: Field declarations.
Arguments: *none*
Semantics: Something along the lines[34] of 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 many fields, but most use cases require only a small subset of them. Because hints do not modify the binary compatibility, some clients are likely to define the fields to be distributed or even on-demand.

Note that this will increase both the memory footprint[35] 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[36]. 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. Hence, the presence of distributed, on-demand or ignored fields may require objects to carry a pointer to their storage pool, which may eliminate the cache savings completely[37].

5.2.2 OnDemand

Can be used on: Field declarations.
Arguments: *none*
Semantics: 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. In prior versions of SKilL, this hint was called *lazy*.

5.2.3 Hide

Can be used on: Field declarations.
Arguments: *none*
Semantics: The generated code will not produce a public API for the field declaration. Hidden fields can be made accessible by views. Further, `hide` implies `onDemand`, because the additional costs in case of reflective access are insignificant.

5.2.4 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.
Semantics: At generation time, the binding generator needs to know tool names that are owned by the generated binding in order to hard-link this behaviour. Arguments are lists of names in order to allow separating 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.

[34] The actual implementation depends on the used generator and may in fact be more like a list of arrays. In fact, the total memory consumption can even be lower compared to the regular implementation, but will likely result in additional field access cost.

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

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

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

Implementation shall treat all types that are not owned by the tool as `ReadOnly`.

Listing 27: Owner Example

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

/* A GraphOwner modifies graphs. ColorOwner modifies
   colored graphs. */
!owner(ColorOwner, GraphOwner)
Node {
  Color color;
  set<Node> edges;
}
```

5.2.5 Provider

Can be used on: Type or field declarations.
Arguments: Names of the tools that provide the target field.
Semantics: If used on type declarations, it extends to all field declarations inside of it and all sub-types. If used on a field declaration the provider is changed to the argument tool for this field. Providers cannot leave their fields partial.

Listing 28: 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;
}
```

5.2.6 Remove Restrictions

Can be used on: Any declaration.
Arguments: Optional restriction names. If none is supplied, all restrictions are removed. If ”unknown” is specified, only unknown restrictions are removed.
Semantics: Removes restrictions obtained from a file instead of keeping them and adhering to them. This Restriction is used to deal with decisions made in the past. 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.

Listing 29: Remove Restriction Example

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

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

5.2.7 Constant Mutator

Can be used on: Constants
Arguments: Range of accepted values (inclusive).
Semantics: 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*. Obviously, appending is not possible if a constant has changed. A binding shall silently fallback to write.

Listing 30: Constant Mutator Example

```
@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;
}
```

5.2.8 Flat

Can be used on: A type with a single field that has neither sub nor super types.
Arguments: *none*
Semantics: Indicates that the type definition shall be hidden in the exported API. This can be used to share containers between multiple entities.

Listing 31: Flat Matrix Example

```
/**
 A matrix definition
 */
Matrix {
```

```
  /** this field should feel like f32[4][4] */
  innerMatrix[4] innerMatrix;
}

/** only used inside of Matrix */
!flat
innerMatrix {
  f32[4] data;
}
```

5.2.9 Unique

Can be used on: Type declarations.
Arguments: *none*
Semantics: Serialization shall unify objects with exactly the same serialized form. In combination with the `@unique`-restriction, no error shall be reported if a duplicate is encountered. This will increase the runtime complexity of the serialization phase.

5.2.10 Pure

Can be used on: Type declarations.
Arguments: *none*
Semantics: 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 is the string pool. An equivalent, in terms of API observable behaviour, would look as follows:

Listing 32: User Strings

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

5.2.11 Monotone

Can be used on: Base type declarations.
Arguments: *none*
Semantics: New instances can be added to the state while existing ones will not be modified. Bindings may use this information to optimize the organization of data and to omit checks. 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.
Arguments: *none*
Semantics: The generated code is unable to modify instances of the respective type.

This hint can be used to provide a consistent API while preventing logical errors, such as modifying data from a previous stage of computation. The hint expands to all subtypes[38]. `ReadOnly` implies `Monotone`. If a `ReadOnly` field is partial, an error has to be thrown while reading the file.

5.2.13 Ignore

Can be used on: Any declaration.
Arguments: *none*
Semantics: 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 fields that 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. Further, `ignore` overrides any other hint and a binding may ignore any restriction applying to an ignored field.

5.2.14 Pragma

Can be used on: Anything.
Arguments: *Implementation-dependent.*
Semantics: 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.

[38]This is because for $A <: B$, all B instances are As as well. If A were not `ReadOnly`, modifying an A would directly modify a B. If B were not `ReadOnly`, modifying a B might modifying an A.

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

Ovid in *Metamorphoses, XV. 165*

Part II
Binary File Format

6 Impact of changing Specifications

Most people who 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 cannot know the exact specification of any given file format.

Let us think about a very simple tool description:

Listing 33: ToolDescription.skill

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

This description can be used by a toolchain 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 from the internet. Thus, adding e.g. the location of a repository containing the tool can be used by a toolchain driver to run tools that have not been installed on the target machine before. Hence, a modern feature may turn the specification into:

Listing 34: 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, so 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 provided by SKilL are carefully designed. They are the result of examination of changes in the Bauhaus toolchain [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.

7 Serialization

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[39] and streams. We also make use of a stream concatenation operator $\circ : S \times S \longrightarrow S$.

We will use upper-case letters for types (e.g. A) and lower-case ones 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. The in-memory representation of all instances that ought to be serialized is called a state, denoted by the letter σ. In this section, we will make use of a set theoretic inspired notation. Thus, $\sigma \supset \mathcal{T}$ holds because all types are known to the state, but it may hold arbitrary additional information. For each instance of a type, we assume that the instance is an element of the state as well.

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 $[\![\_]\!]_\tau : \tau \times \mathcal{T} \longrightarrow S$ 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.

7.1 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 can be 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[40], fields using these indices have to be updated.
   - All known restrictions have to be checked.
3. The output stream is created as described below.

7.2 General File Layout

The file layout is optimized for fast appending of new objects. It is further optimized for on-demand and partial loading of existing objects. It also supports 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 uses 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.

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

[40] Indices can change if objects are inserted before an existing object or if objects are deleted. This is caused by the base pool index concept explained in section 7.3

Consequently, 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.

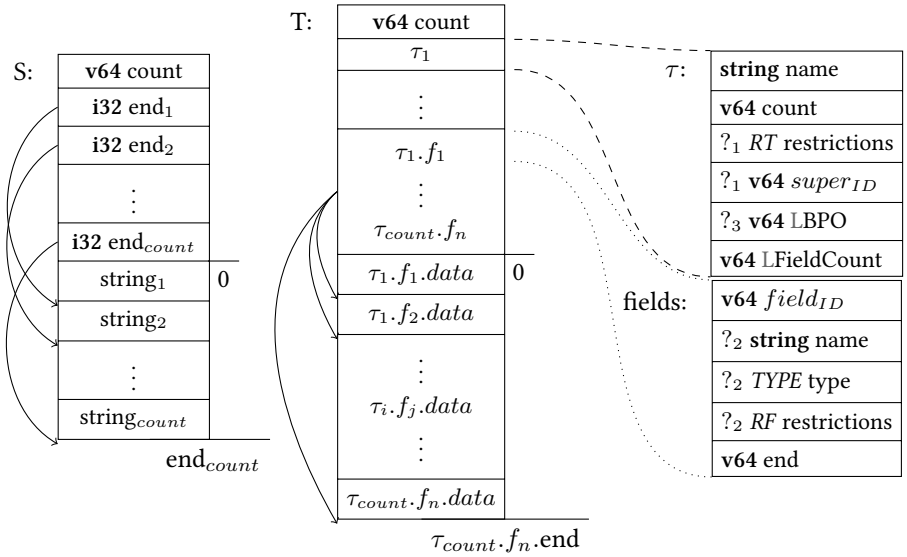

Figure 2: Visualization of the layout of string(S) and type(T) blocks. Type descriptors (τ) and their fields can drop fields in some contexts ($?_i$). Fields prefixed with a gray L contain information only relevant for the local block. The data chunks of blocks start at the respective **0** and reach to the respective **end**. Arrows indicate end-offsets – type blocks have one per field declaration. Field restrictions have to be placed after the field's type, because their translation may depend on the type. *LFieldCount* many field descriptions are located after LFieldCount – similar to the serialization of arrays.

7.2.1 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[41] of the end of the respective string. The stored offsets split the data following 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 subsequent type blocks. For efficiency[42] 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 and field names have to be parsed, but the interference with potentially many user-strings won't hurt the runtime of store operations.

[41] in bytes

[42] The lower-case conversion during deserialization is optimized away.

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

Further Properties of Type Names Type names may contain $'$ $:'$ characters. They are currently used by `enum` instances and will eventually be used for name spaces.

7.2.2 Layout of a Type Block

The instances of all types are organized into storage pools (see section 7.3) 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 of type declarations. Type declarations form the type hierarchy and contain references to field declarations.

After all type declarations, referenced field declarations appear in the order of their declaration. Field declarations 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 supertype 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). If the LBPO field is not present it can 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 7.2.8.

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 speeding up operations on types and fields.

7.2.3 Type Order

Type order is a partial order that linearizes 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 than 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 tools using different specifications added data to a file.

A stable type order is obtained by 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 the 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.

7.2.4 Effects of On-Demand Deserialization

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

Inside 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 result of this decision is also 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 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 several kilobytes at most.

7.2.5 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 an 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 7.3) are made necessary by this feature as well. The omitted data is marked with $?_1$ and $?_2$ in Fig. 2: Fields marked with $?_1$ only appear in the first block and are left out 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 cannot 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 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.

7.2.6 Partial Fields

A field that lacks instance data or does not comply with 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 with a restriction. Default values are serialized alongside regular field data. Therefore, in this case *missing* field data can be substituted with default values.

7.2.7 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 and other `auto` fields[43]. Field IDs are dense and fields always appear in an ascending order. Field IDs are only relevant to the serialization mechanism and do not correspond to a specification in general. Field IDs are counted for each type individually, i.e. IDs are independent of fields in super types. Note that it is not possible in general to have continuous field IDs that take sub types into account, because appending may require inserting fields to a super type causing at least one of the properties stated above to be violated.

7.2.8 Node Example

Let us assume two tools with two SKilL specification files:

Listing 35: Specification of the Node Producer Tool

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

Listing 36: 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[44], 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’.

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

Adding instances Now let us assume that the first tool runs twice, adding 23/42 in the first run and -1/2 in the second. We won’t run the coloring tool, thus S/T are the result of the first run and S*/T* are the result of the second run[45] (see Fig. 4).

[43] Note on implementations: We rule semantics of all field IDs smaller than 0 implementation defined, however, they must never be written into a file This can be used to map auto fields into an array using a unary minus and non-auto fields into another array using a decrement.

[44] The file can be found in the test suite as `coloredNodes.sf`.

[45] The file can be found in the test suite as `twoNodeBlocks.sf`.

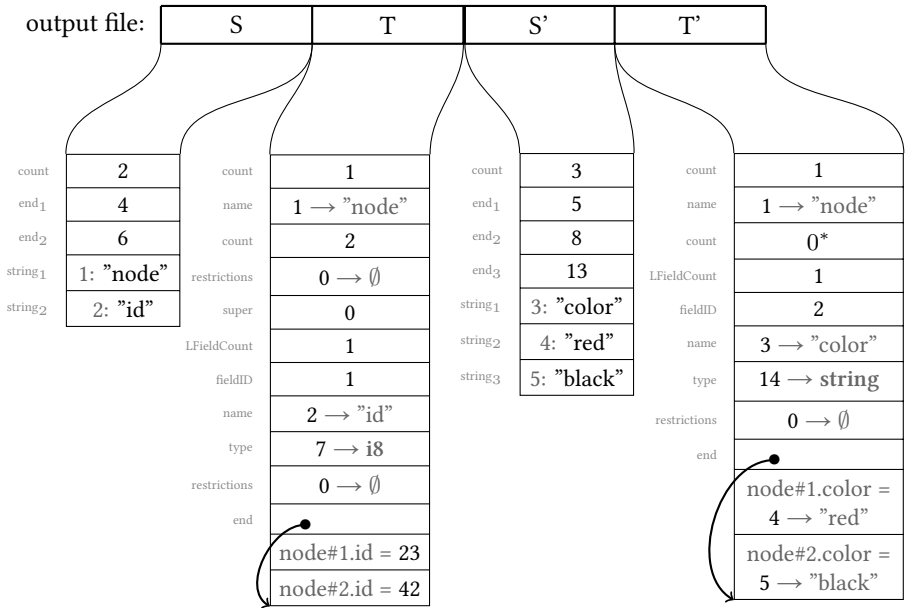

Figure 3: Illustration of the file obtained after running the example toolchain. Gray parts are the interpretations of the respective value and will be explained below. Light gray annotations shall serve as a reminder of the meaning of the contents of the respective field. In section 7.6 there is an example explaining serialization as a process.

7.3 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: $baseTypeName : \mathcal{U} \rightarrow S$, $typeName : \mathcal{U} \rightarrow S$ and $index : \mathcal{U} \cup \{\textbf{string}\} \rightarrow S$.

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.

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 the number of instances it holds. Storage pools with a supertype store the name of the supertype and a BPO. Further, we assume that objects to be serialized have indices such that, for any type, all instances of that type have adjacent indices.

A short example (Fig. 5) illustrates the concept. It contains five types **A**, **B**, **C**, **D**,

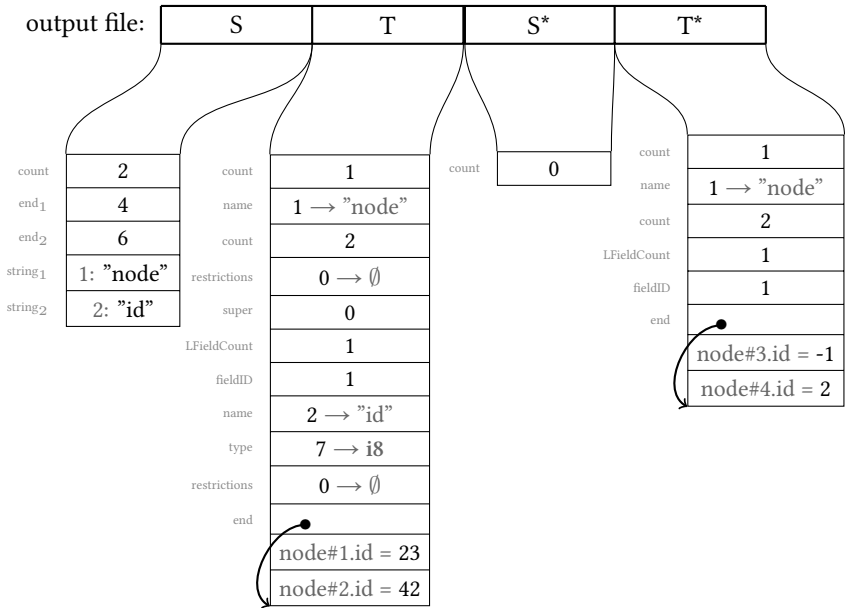

Figure 4: Illustration of the file obtained after running the example toolchain. Gray parts are the interpretations of the respective value and will be explained below. Light gray annotations provide contents descriptions. Note that string blocks (and type blocks) can be empty but must not be omitted.

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 1/5/4, 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. For the sake of readability, the header of the type pool is omitted in the stream part of the picture. Note that field data appears in the order of declaration.

7.3.1 Subtypes

The presence of subtypes requires storage pools to be laid out such that accessing, allocation and type checking can be performed efficiently. To do so, there are several noteworthy invariants of BPOs:

$$first.index - base.first.index = bpo$$
$$first.index - 1 + count = last.index$$

Furthermore, if there is exactly one subtype, then $count = sub.bpo + sub.count$ and for each subtype, the following equation holds: $count \leq sub.bpo + sub.count$

7.3.2 Appending Fields to existing Pools

Appending a field to a pool does not modify existing instances. The respective type block will have a type declaration with a field declaration of the added field. Fields can be appended to a file without further constraints because all required information can be obtained by an inexpensive lookup.

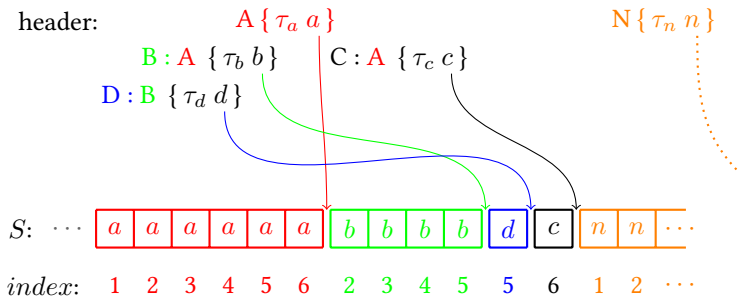

Figure 5: Field data of several pools stored in a type block. Arrows indicate the values of data fields of the respective field description in the header of the type block. S shows a part of the data chunk of a type block. LBPOs of B/C/D are 1/5/4.

7.3.3 Appending Objects to Existing Pools

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 `adc`. T1 contains the full type information of types A,B and C. There are 6 A instances, 3 B instances (with LBPO=2) and 1 C instance (with LBPO=5). T2 contains only the type name of A and B (LBPO=0) and field data for 4 instances each. Additionally, there is a new type D (with LBPO=2) and D-field data for two instances. T3 contains field data for A, B (LBPO=1), D (LBPO=1) and C (LBPO=2) objects. The in-memory representation after loading these objects is expected[46] to be either `aabbbcbbddadc`, `aaaccbbbbbddd` or `aaabbbbbddddcc`[47].

7.3.4 Omission of Data

Empty storage pools, i.e. pools with $count = 0$ must not be serialized. Fields that use only default values must not serialize any field data. This behaviour is required because fields that contain null-Pointers only may cause files to be inconsistent.

These rules address an issue of a prior version of SKilL where significant amounts of type specification had to be stored in empty files. Simply omitting types without instances is not enough because other types may refer to them. Furthermore, the default restriction mechanism is used to further compress files with little entropy.

Note on Implementations Marking default-only fields and empty pools should happen before collecting strings and after collecting other instances. Otherwise, the string block will get polluted with non-existent type and field names. Further, files resulting from disobeying this rule shall be readable and produce correct results.

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

[47] The file can be found in the test suite as `localBasePoolOffset.sf`.

7.4 Serialization of Field Data

In this section, we want to describe the serialization of individual fields using the function $[\![\_]\!]_\tau$. 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 into streams using the `v64` encoding. We assume further that 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[48] non-`auto` field of type t, then $[\![f]\!]_t$ is defined as[49]

- $\forall t \in \mathcal{U} \cup \{\textbf{string}\}.[\![f]\!]_t = \begin{cases} \texttt{0x00}, & f = \texttt{NULL} \\ [\![index(f)]\!]_{v64} & else \end{cases}$
- $[\![f]\!]_{\textbf{annotation}} = \begin{cases} \texttt{0x00 0x00}, & f = \texttt{NULL} \\ [\![1 + storagePoolIndex(t_f)]\!]_{v64} \circ [\![index(f)]\!]_{v64} & else \end{cases}$
- $[\![\top]\!]_{\textbf{bool}} = [\![\varepsilon x.x \neq 0]\!]_{i8}$[50]
- $[\![\bot]\!]_{\textbf{bool}} = \texttt{0x00}$
- $\forall t \in \mathcal{I} \setminus \{\textbf{v64}\}.[\![f]\!]_t = f$
- $[\![f]\!]_{\textbf{v64}} = encode(f)$[51]
- $[\![f]\!]_{\textbf{f32}} = [\![f]\!]_{\textbf{i32}}, [\![f]\!]_{\textbf{f64}} = [\![f]\!]_{\textbf{i64}}$[52]
- $\forall g \in \mathcal{T}, n \in \mathbb{N}^+.t = g\texttt{[}n\texttt{]} \implies [\![f]\!]_t = \bigcirc_{i=0}^{n-1} [\![f_i]\!]_g$
- $\forall g \in \mathcal{B}, n = size(f), t \in \{g\texttt{[]}, \texttt{set<}g\texttt{>}, \texttt{list<}g\texttt{>}\}.[\![f]\!]_t = [\![n]\!]_{v64} \bigcirc_{i=0}^{n-1} [\![f_i]\!]_g$
- $\forall r, s, t \in \mathcal{T}.[\![\_]\!]_{map<r,s,t>} = [\![\_]\!]_{map<r,map<s,t>>}$
- $\forall k, v \in \mathcal{T}, n = size(f), t = \texttt{map<}k, v\texttt{>}.[\![f]\!]_t = [\![n]\!]_{v64} \bigcirc_{i=0}^{n-1} [\![f.k_i]\!]_k [\![f[k_i]]\!]_v$
- $[\![rs]\!]_{RT}/[\![rs]\!]_{RF} =$ 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.

- $[\![t]\!]_{type} = \begin{cases} [\![id]\!]_{v64} \circ [\![val]\!]_t & id \in [0, 4] \\ [\![id]\!]_{v64} & id \in [5, 14] \\ \texttt{0x0F} \circ [\![i]\!]_{v64} \circ [\![T]\!]_{type} & t = T[i] \\ \texttt{0x11} \circ [\![T]\!]_{type} & t = T[] \\ \texttt{0x12} \circ [\![T]\!]_{type} & t = list\langle T \rangle \\ \texttt{0x13} \circ [\![T]\!]_{type} & t = set\langle T \rangle \\ \texttt{0x14} \circ [\![K]\!]_{type} \circ [\![V]\!]_{type} & t = map\langle K, V \rangle \\ [\![32 + storagePoolIndex(t)]\!]_{v64} & t \in \mathcal{U} \end{cases}$

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

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

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

[51] With encode as defined in listing 39.

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

*id*s of restrictions and types are listed in appendix G. The holes are intentional and enable future built-in types without breaking the file format.

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

Note Container types take arbitrary other types as arguments. This is different from the specification language and enables future relaxations of the type system that may for instance allow two dimensional arrays, e.g. to represent a matrix in-place.

7.5 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.

7.6 Age Example

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

Listing 37: Age File Format

```
Age {
  /* people have a small positive age, but maybe they
     will start to live longer in the future, who knows
     */
  @min(0)
  v64 age;
}
```

Now we want to store two objects with *age* values 1 and 28. So, 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:

$$Age = \{[age : 1], [age : 28]\}$$

Now we have to create storage pools. We start with the creation of type information. After the collection of instances, the age storage pool looks something like this:

$$[name : \text{"}age\text{"}, restr. : \emptyset, count : 2, super : 0, fields[$$
$$1 : [restr. : \text{"}min(0)\text{"}, t : \texttt{v64}, n : \text{"}age\text{"}, [1, 28]]]$$

Now we have to check all restrictions. The only relevant restriction of field age is obviously fulfilled.

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 until now), so we create a string pool[53] as well:

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

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

Now we can start to encode the pools:

1. write the string block:

   $[\![[1 : \text{"}age\text{"}]]\!] = [\![1]\!]_{v64} \circ offset(\text{"}age\text{"}) \circ [\![\text{"}age\text{"}]\!]$
   $= [\![1]\!]_{v64} \circ [\![3]\!]_{i32} \circ \texttt{61 67 65}$
   $= \texttt{01 00 00 00 03 61 67 65}$

2. write the type block:

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

   $[\![[name : \text{"}age\text{"}, count : 2, restr. : \emptyset, super : 0, \cdots]\!]$
   $= [\![1]\!]_{v64} \circ [\![\text{"}age\text{"}]\!] \circ [\![2]\!]_{v64} \circ [\![\emptyset]\!]_{RT[]} \circ [\![0]\!]_{v64}$
   $= 1 \circ index(\text{"}age\text{"}) \circ 2 \circ [\![size(\emptyset)]\!]_{v64} \circ 0$
   $= 1 \circ 1 \circ 2 \circ 0 \circ 0 = \texttt{01 01 02 00 00}$

   Now we can continue with the non-data part of the field:

   $[\![[\cdots fields[1 : [restr. : \text{"}min(0)\text{"}, t : \texttt{v64}, n : \text{"}age\text{"}, [1, -1]]]]\!]$
   $= [\![size(fields[\cdots])]\!] \circ \cdots$
   $= \texttt{01} \circ [\![1 : [n : \text{"}age\text{"}, t : \texttt{v64}, restr. : \text{"}min(0)\text{"} \ldots]\!]$
   $= \texttt{01} \circ [\![1]\!]_{v64} \circ [\![\text{"}age\text{"}]\!] \circ [\![\texttt{v64}]\!]_{TYPE} \circ [\![\text{"}min(0)\text{"}]\!]_{RF[]} \ldots$
   $= \texttt{01 01 01 0B} \circ [\![\text{"}min(0)\text{"}]\!]_{RF[]} \ldots$
   $= \texttt{01 01 01 0B} \circ [\![1]\!]_{v64} \circ [\![\text{"}min(0)\text{"}]\!]_{RF} \ldots$
   $= \texttt{01 01 01 0B} \circ [\![1]\!]_{v64} \circ [\![3]\!]_{v64} \circ [\![0]\!]_{v64} \circ [\![2^{63-1}]\!]_{v64} \ldots$
   $= \texttt{01 01 01 0B 01 03 00 FF FF FF FF FF FF FF 7F} \ldots$

   And, finally, we can serialize data:

   $= \cdots [\![offset([1, 28])]\!] \circ [\![[1, 28]]\!]$
   $= \cdots \circ offset(\text{"}age.age\text{"}) \circ [\![[1, 28]]\!]$
   $= \cdots \circ \texttt{02}[\![1]\!]_{v64} \circ [\![28]\!]_{v64}$
   $= \cdots \texttt{02 01 1C}$

   The type block is now serialized to the stream:

   ```
   01 01 01 0B 01 03 00 FF FF FF FF FF FF FF 7F 02 01 1C
   ```

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[54]:

   ```
   01 00 00 00 03 61 67 65 01 01 02 00 00 01 01 01 0B 01 03 00
   FF FF FF FF FF FF FF FF 7F 02 01 1C
   ```

Please note that although it may seem that a lot of space is used for the serialization of type information, this is not the case, as SKilL is designed for thousands to millions of instances of a type. Deserialization of this stream is explained in section 8.1.

[54] This is part of the test suite as `age.sf`.

7.7 Map Example

In this short section, we will give an example of 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 map's 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 the 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`

8 Deserialization

Deserialization is mostly straightforward. 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. unless the end of file has been reached, goto step 1

8.1 Age Example

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 7.6:

```
01 00 00 00  03 61 67 65  01 01 02 00  00 01 01 01  0B 01 03 00
FF FF FF FF  FF FF FF FF  7F 02 01 1C
```

1. deserialzation of a string block:

   $d(0100000003616765010102\cdots)$
   $\rightarrow$a string block starts with a v64 indicating the number of strings stored inside

   $d(01)d(00000003616765010102\cdots)$
   $\rightarrow$we got one string

   $d(01)d(00000003)d(616765010102\cdots)$
   $\rightarrow$the next string has 3 bytes, the block ends in 3 bytes

   $string[1 : d(616765)]d(0101\cdots)$
   $\rightarrow$build the string pool with the first string block

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

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

   $d(01)d(010200000101010B010300\cdots)$
   $\rightarrow$there is one type definition in this block; read its name

   $d(01)d(0200000101010B010300\cdots)$
   $= "date"d(02)d(00)d(00)0101010B010300\cdots)$
   $\rightarrow$we do not know the type "date" yet (in terms of processing the file), so we expect count, restrictions, super type information and field declarations

   $[name : "age", count : 2, restr. : \emptyset, super : 0]d(01)d(01010B010300\cdots)$
   $\rightarrow$Date has no super type, thus the next field is not a local BPO and we can read until the number of fields. $\rightarrow$We read all types of the block, so we continue with field declarations – we have one field for type age.

   $[name : "age", super : 0, count : 2, restr. : \emptyset, fields[1 : \_]]$
   $d(01)d(010B010300\cdots)$
   $\rightarrow$We do not know age.field#1 yet. Therefore, the next pieces of information are name, type, field restrictions and offset.

   $[name : "age", super : 0, count : 2, restr. : \emptyset, fields[1 : \_]]$
   $d(01)d(0B)d(01)d(0300\cdots)$
   $\rightarrow$Name is age, type is v64 and there is one restriction.

   $[name : "age", super : 0, count : 2, restr. : \emptyset, fields[1 : [name : "age", type : v64, restr. : \_, offset : \_]]$
   $d(03)d(00FFFFFFFFFFFFFFFF7F02011C)$
   $\rightarrow$Found range, consume borders
   $d(03)d(00)d(FFFFFFFFFFFFFFFF7F)d(02011C)$
   $\rightarrow$Range [0; v64.max]

$[name : "age", super : 0, count : 2, restr. : \emptyset, fields[1 : [name : "age", type : v64, restr. : min(0), offset : \_]]$
$d(02)d(011C)$
⟶offset is 2, so we are done with all field declarations

3. construction of date objects:

   $Age = \{[age : \_], [age : \_]\}d(011C)$
   ⟶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

   $Age = \{[age : 1], [age : \_]\}d(1C)$
   ⟶the first field is a 1

   $Age = \{[age : 1], [age : 28]\}$
   ⟶the second field is a 28; 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 7.6.

Part III
Future Work

> 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*

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 by interfaces and hints or restrictions. We will also evaluate possible ways to support type casts in order to allow using language-specific types, such as bit-fields.

Further research has to be done in the area of restrictions. Their impact on performance and usability remains widely uncertain. Thus, they should be evaluated in a real toolchain. We will introduce a general assertion restriction which can be used to assert per instance invariants with sufficiently powerful expressions.

We will further evaluate means of supporting common high-level specification tasks, such as creating effective views for individual tools of a toolchain. This can either be in the form of a SKilL specification editor or an extended specification language. A general-purpose viewer and editor for SKilL files should be implemented nonetheless.

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?

Numerical limits, as described in appendix F, will be dropped as all widely used programming languages support 64-bit index types.

The concept of views introduced in this version may be extended in the future. Likewise, the concept of partial fields and ownership requires thorough evaluation.

Part IV
Appendix

A Full Grammar of the Specification Language

⟨*file*⟩	::=	⟨*header*⟩ ⟨*declaration*⟩*
⟨*header*⟩	::=	⟨*head-comment*⟩* ⟨*include*⟩*
⟨*head-comment*⟩	::=	'#' (~[\n])* '\n'
⟨*include*⟩	::=	(`include`|`with`) ⟨*string*⟩+
⟨*declaration*⟩	::=	⟨*user-type*⟩
	|	⟨*interface-type*⟩
	|	⟨*enum-type*⟩
	|	⟨*typedef*⟩
⟨*user-type*⟩	::=	⟨*description*⟩ ⟨*ID*⟩ ((':'|`with`|`extends`) ⟨*ID*⟩)* '{' ⟨*field*⟩* '}'
⟨*interface-type*⟩	::=	⟨*comment*⟩? `interface` ⟨*ID*⟩ ((':'|`with`|`extends`) ⟨*ID*⟩)* '{' ⟨*field*⟩* '}'
⟨*enum-type*⟩	::=	⟨*comment*⟩? `enum` ⟨*ID*⟩ '{' ⟨*ID*⟩ (',' ⟨*ID*⟩)* ';' ⟨*field*⟩* '}'
⟨*typedef*⟩	::=	⟨*comment*⟩? `typedef` ⟨*ID*⟩ (⟨*restriction*⟩|⟨*hint*⟩)* ⟨*type*⟩ ';'
⟨*field*⟩	::=	⟨*description*⟩ (⟨*constant*⟩|⟨*data*⟩|⟨*view*⟩|⟨*custom*⟩) ';'
⟨*constant*⟩	::=	`const` ⟨*type*⟩ ⟨*ID*⟩ '=' ⟨*int*⟩
⟨*data*⟩	::=	(`auto`)? ⟨*type*⟩ ⟨*ID*⟩
⟨*view*⟩	::=	`view` (⟨*ID*⟩ '.')? ⟨*ID*⟩ 'as' ⟨*type*⟩ ⟨*ID*⟩
⟨*custom*⟩	::=	`custom` ⟨*ID*⟩ ('!' ⟨*ID*⟩ (⟨*STRING*⟩ ('(' ⟨*STRING*⟩* ')')?)?)* ⟨*STRING*⟩ ⟨*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*⟩|'\n'])*)*
⟨*comment-prefix*⟩	::= '\n' ⟨*whitespace*⟩ ('*' ⟨*whitespace*⟩)?
⟨*comment-TAG*⟩	::= '@' [a-z]+
⟨*restriction*⟩	::= '@' ⟨*ID*⟩ ('(' (⟨*argument*⟩ (',' ⟨*argument*⟩)*)?')')?
⟨*hint*⟩	::= '!' ⟨*ID*⟩ ('(' (⟨*argument*⟩ (',' ⟨*argument*⟩)*)? ')')?
⟨*argument*⟩	::= ⟨*FLOAT*⟩ | ⟨*INT*⟩ | ⟨*STRING*⟩ | (⟨*ID*⟩ (('.'|'::') ⟨*ID*⟩)*)

Table 3: SKilL Specification Descriptions

B Full Grammar of the Binary File Format

The notation is explained further in chapter 7. Additionally, we use small gothic letters for blocks, to avoid collisions. This is a synopsis of chapters 7 and 5.1. Type constants and restriction parameters are given in appendix G.

$$S = \mathfrak{s}_1 \circ \mathfrak{t}_1 \circ \cdots \mathfrak{s}_n \circ \mathfrak{t}_n$$
$$\mathfrak{s} = [\![n]\!]_{v64} \circ [\![end_1]\!]_{i32} \cdots [\![end_n]\!]_{i32} \circ [\![string_1]\!]_{utf8[]} \cdots [\![string_n]\!]_{utf8[]}$$
$$\mathfrak{t} = [\![n]\!]_{v64} \circ [\![t_1]\!]_{\mathbb{T}} \cdots [\![t_n]\!]_{\mathbb{T}} \circ \bigcirc_{t_i \in \mathfrak{t}} \bigcirc_{f_j \in t_i} [\![t_i.f_j]\!]_{\mathbb{F}} \circ \bigcirc_{t_i \in \mathfrak{t}} \bigcirc_{f_j \in t_i} \bigcirc_{x \in t_i} [\![x.f_j]\!]_{t_i.f_j.type}$$

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

$$[\![f]\!]_{\mathbb{F}} = [\![ID]\!]_{v64} \circ [\![f]\!]_{?2} \circ [\![end]\!]_{v64}$$
$$[\![t]\!]_{?2} = \begin{cases} [\![name]\!]_{string} \circ [\![t]\!]_{TYPE} \circ [\![rs]\!]_{RF[]} & \text{first occurrence of } f \text{ in } S \\ \varepsilon & else \end{cases}$$
$$\forall t \in \mathcal{U} \cup \{\textbf{string}\}.[\![f]\!]_t = \begin{cases} \texttt{0x00}, & f = \texttt{NULL} \\ [\![index(f)]\!]_{v64} & else \end{cases}$$
$$[\![f]\!]_{\textbf{annotation}} = \begin{cases} \texttt{0x00 0x00} & f = \texttt{NULL} \\ [\![1 + storagePoolIndex(t_f)]\!]_{v64} \circ [\![index(f)]\!]_{v64} & else \end{cases}$$
$$[\![\top]\!]_{\textbf{bool}} = [\![\varepsilon x.x \neq 0]\!]_{i8},\ [\![\bot]\!]_{\textbf{bool}} = \texttt{0x00}$$
$$\forall t \in \mathcal{I} \setminus \{\textbf{v64}\}.[\![f]\!]_t = f$$
$$[\![f]\!]_{\textbf{v64}} = [\![v, 0]\!]_{v64}$$
$$[\![v, i]\!]_{v64} = \begin{cases} [\![\texttt{0x80}|v]\!]_{i8} \circ [\![v >> 7, i+1]\!]_{v64} & v >> 7 > 0 \vee i \leq 8 \\ [\![v]\!]_{i8} & else \end{cases}$$
$$[\![f]\!]_{\textbf{f32}} = [\![f]\!]_{\textbf{i32}},\ [\![f]\!]_{\textbf{f64}} = [\![f]\!]_{\textbf{i64}}$$
$$\forall g \in \mathcal{T}, n \in \mathbb{N}^+.t = g\texttt{[}n\texttt{]} \implies [\![f]\!]_t = \bigcirc_{i=0}^{n-1} [\![f_i]\!]_g$$

$$\forall g \in \mathcal{B}, n = size(f), t \in \{g\texttt{[]}, \texttt{set<}g\texttt{>}, \texttt{list<}g\texttt{>}\}.[\![f]\!]_t = [\![n]\!]_{v64} \bigcirc_{i=0}^{n-1} [\![f_i]\!]_g$$
$$\forall r, s, t \in \mathcal{T}.[\![\_]\!]_{map<r,s,t>} = [\![\_]\!]_{map<r,map<s,t>>}$$
$$\forall k, v \in \mathcal{T}, n = size(f), t = \texttt{map<}k, v\texttt{>}.[\![f]\!]_t = [\![n]\!]_{v64} \bigcirc_{i=0}^{n-1} [\![f.k_i]\!]_k [\![f[k_i]]\!]_v$$
$$[\![rs]\!]_{RT} / [\![rs]\!]_{RF} = [\![ID]\!]_{v64} \circ parameters(rs)$$
$$[\![t]\!]_{type} = \begin{cases} [\![id]\!]_{v64} \circ [\![val]\!]_t & id \in [0, 4] \\ [\![id]\!]_{v64} & id \in [5, 14] \\ \texttt{0x0F} \circ [\![i]\!]_{v64} \circ [\![T]\!]_{type} & t = T[i] \\ \texttt{0x11} \circ [\![T]\!]_{type} & t = T[] \\ \texttt{0x12} \circ [\![T]\!]_{type} & t = list\langle T \rangle \\ \texttt{0x13} \circ [\![T]\!]_{type} & t = set\langle T \rangle \\ \texttt{0x14} \circ [\![K]\!]_{type} \circ [\![V]\!]_{type} & t = map\langle K, V \rangle \\ [\![32 + storagePoolIndex(t)]\!]_{v64} & t \in \mathcal{U} \end{cases}$$

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 bits 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 38: 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.

Listing 39: 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

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 results of file corruption, format change or bugs in a language binding.

Serialization

- If a restriction check fails, an error must be reported before creating any observable output and without performing observable[55] modifications to an input stream or the state itself.

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[56].
- If an index into a pool is invalid[57], 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.
- 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[58] 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

[55] The state of on-demand reading is not observable in this context.

[56] This is less restrictive than claiming that no offset may point after EOF. This way, partially lost data can still be processed.

[57] because it is larger than the last index/size of the pool

[58] I.e. after a subtype but not inside another subtype

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 field differs by an auto modifier, an error must be reported before producing any observable results.
- If a serialized value violates a restriction or the invariant of a type,[59] an error must be reported as soon as this fact can be observed[60].
- If a restriction cannot 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.
- If a known type has a super type different from the one in the specification, an error has to be reported including the specified and the actual super type as well as the type itself.

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 two parts. The core level can be implemented quickly and suffices for many SKilL-related tasks. Further, core level bindings will produce valid files in almost all usage scenarios. Invalid files can only be produced by type errors that remain undetected because the type system of a core-level binding is strictly weaker than that of a full implementation.

Some properties of SKilL are considered features whose implementation is almost orthogonal.

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 with implementations of the previous standard showed that many designs for-core level[61] bindings cannot be extended to full-featured SKilL bindings.

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 features implemented by a specific language binding (generator).

- Core
  - treatment of all SKilL types
  - reading of valid files
  - writing of files

[59] Including sets containing multiple similar objects.
[60] This may in fact be never if the field has an onDemand hint and is not used.
[61] respective to the old standard

  - implementation of restrictions: Singleton, Default, NonNull
  - means of SKilL state management
  - a public reflective interface allowing fully generic file manipulation

- Features

auto: management of `auto` fields

append: appending to files

conversion: implementation of legal type conversions and prevention of appends if conversions occurred

customs: treatment of `custom` fields

documented: language-dependent treatment of comments, e.g. integration into doxygen or javadoc

enums: implementation of enumerations if target language supports them

escaped: name mangling to allow usage of language keywords or illegal characters (unicode) in specification files without making a language binding impossible

hint: implementation of hints

interfaces: implementation of interfaces, if target language supports them

lazy: implementation of on-demand deserialization

limits: support for files exceeding the guaranteed numerical limit range[62]

multi-state: safe behaviour of interleaving states

restricted: implementation of all restrictions

safe: implementation of all specified Error reporting

typedefs: implementation of typedefs if target language supports them

view: treatment of views

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 cannot deal with arrays of a size larger than about 2^{31}. Therefore, we establish the following rule:

(De-)serialization of a file with an array of more than 2^{30} elements or a type with more than 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 an array of characters. Therefore, their length is expected to not exceed 2^{30} characters as well.

[62] 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 7.4) and restriction IDs (see sections 5.1 and 5.2). Restrictions with even IDs can be ignored if unknown.

Type IDs:

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

Restriction IDs for Types:

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

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	ε
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.

[BKEdN] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. Yaml ain't markup language.

[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.

[Fel13] Timm Felden. The SKilL Language. Technical Report 2013/06, University of Stuttgart, September 2013.

[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.

[Har14] Fabian Harth. Plattform- und sprachunabhängige Serialisierung mit SKilL. Diploma thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, November 2014.

[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.

[Prz14] Dennis Przytarski. Performance-Evaluation einer sprach- und plattformunabhängigen Serialisierungssprache. Bachelorarbeit: Universität Stuttgart, Institut für Softwaretechnologie, Programmiersprachen und Übersetzerbau, Juni 2014.

[Rot15] Jonathan Roth. Reduktion des Speicherverbrauchs generierter SKilL-Zustände. Master thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, May 2015.

[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.

[SSV13] Fabian Schomm, Florian Stahl, and Gottfried Vossen. Marketplaces for data: An initial survey. *SIGMOD Rec.*, 42(1):15–26, May 2013.

[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.

[Ung14] Wladislaw Ungur. Nutzbarkeitsevaluation einer sprach- und plattformunabhängigen Serialisierungssprache. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Juli 2014.

[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.