CZECH TECHNICAL UNIVERSITY IN PRAGUE
Faculty of Electrical Engineering

BACHELOR’S THESIS

Extending the Kaitai Struct parser generator to support serialization

Petr Pučil

Supervisor: Ing. Michal Sojka, Ph.D.

Field of study: Open Informatics

Subfield: Internet of Things

May 2024

BACHELOR‘S THESIS ASSIGNMENT

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

I. Personal and study details

Student's name: **Pučil Petr** Personal ID number: **507271**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Measurement**

Study program: **Open Informatics**

Specialisation: **Internet of Things**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Extending the Kaitai Struct parser generator to support serialization

Bachelor’s thesis title in Czech:

Rozšíření generátoru parserů Kaitai Struct o podporu serializace

Guidelines:

Description
Kaitai Struct is an open source tool for dealing with binary formats. It introduces a declarative domain-specific language for describing the structure of arbitrary binary formats.
So far, Kaitai Struct only supports parsing. You can extract data from binary files created by other programs. However, in many cases, the opposite direction is also needed, i.e. to modify the data in the binary files or to create new files, so-called serialization. The goal is to implement serialization in Kaitai Struct for Java. Requirements:

1. Study the principles of serialization of binary formats and compare the serialization approaches of existing tools
2. Analyze the existing state and capabilities of the Kaitai Struct parser generator
3. Design an extension to Kaitai Struct to include a new module that would allow serialization of binary formats based on Kaitai Struct YAML specifications
4. Implement the proposed module for Java
5. Design an efficient way to test serialization, create a test infrastructure.
6. Verify the functionality of the serialization module.

Bibliography / sources:

1. VIOTTI, Juan Cruz, and Mital KINDERKHEDIA. A Survey of JSON-compatible Binary Serialization Specifications [online]. arXiv preprint, 2022-01-10: https://doi.org/10.48550/arXiv.2201.02089
2. MORSCHEL, Lea. Efficient Message Serialization for Inter-Service Communication in dCache [online]. Course work. Wedel: University of Applied Sciences Wedel, 2019: https://doi.org/10.3204/PUBDB-2020-01022
3. Kaitai Struct documentation: https://doc.kaitai.io/
4. Java documentation: https://docs.oracle.com/javase/7/docs/api/

Name and workplace of bachelor’s thesis supervisor:

Ing. Michal Sojka, Ph.D. Embedded Systems CIIRC

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **17.01.2024** Deadline for bachelor thesis submission: **24.05.2024**

Assignment valid until:
by the end of summer semester 2024/2025

Ing. Michal Sojka, Ph.D. Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Acknowledgements

First and foremost, I would like to thank my loving family for the unconditional support they have given me throughout my life in every way possible.

I would also like to thank my supervisor Michal Sojka for his great advice throughout the writing of the thesis.

I would like to thank the founder of the Kaitai project, Mikhail Yakshin from Microsoft, for the trust he placed in me by accepting me into the Kaitai development team and subsequently granting me the rights of administrator and project co-owner. I also appreciate his continuous help in the development of Kaitai Struct.

Special thanks to Armijn Hemel from the Dutch consulting firm Tjaldur Software Governance Solutions, who has been supporting my work and ideas over the last years. He is always willing to give me help and advice when I am not sure if I am taking the right steps, not only in Kaitai Struct development. The speed of response to my questions is admirable. “He gives twice who gives quickly.”

The project “Serialization in Kaitai Struct” has received a grant from the NLnet Foundation. Many thanks to its director Michiel Leenars for his support of Kaitai Struct, which continues with another grant project.

Declaration

I hereby declare that the submitted thesis is exclusively my own work and that I have listed all used information sources in accordance with the Methodological Guideline on Ethical Principles for College Final Work Preparation.

Cesky Krumlov, May 1, 2024

Abstract

Kaitai Struct (KS) is a tool for working with binary formats. It introduces a declarative domain-specific language Kaitai Struct YAML (.ksy) for describing the structure of arbitrary binary formats. Based on any specification, KS can automatically generate a ready-to-use parsing module in one of 11 programming languages. Until now, KS only allowed to parse data from binary files to an object tree. However, in many cases, the opposite direction (serialization) is also needed, i.e., to modify the data in binary files or to create new ones. This can be used to convert files between formats, to transfer data between computers using a transport protocol, or for fuzzing, useful for detecting bugs in parsers. This work adds serialization support for Java to the KS project. This involves extending the compiler, adding support for serialization in the runtime library, and building an automated testing infrastructure for serialization. The result is a serialization module that works for almost all KSY format specifications in the official format gallery, of which there are over 180. To ensure that the implementation works in identified edge cases, 59 unit tests were added.

Keywords

Kaitai Struct, parser generators, serialization, parsing, binary formats

---

Abstrakt

Kaitai Struct (KS) je nástroj pro práci s binárními formáty. Zavádí deklarativní doménově specifický jazyk Kaitai Struct YAML (.ksy) pro popis struktury libovolných binárních formátů. Na základě jakékoliv specifikace dokáže KS automaticky vygenerovat hotový modul pro parsování v jednom z 11 programovacích jazyků. Dosud KS umožňoval pouze parsovat data z binárních souborů do stromu objektů. V mnoha případech je však potřeba i opačný směr (serializace), tj. úprava dat v binárním souboru nebo vytvoření nového. To lze použít k převodu souborů mezi formáty, k přenosu dat mezi počítači pomocí transportního protokolu nebo k fuzzingu, což je užitečné pro detekci chyb v parserech. Tato práce přidává do projektu KS podporu serializace pro Javu. To zahrnuje rozšíření kompilátoru, přidání podpory serializace do běhové knihovny a vybudování automatizované testovací infrastruktury pro serializaci. Výsledkem je serializační modul, který funguje pro téměř všechny formátové specifikace KSY v oficiální galerii formátů, kterých je více než 180. Aby bylo zajištěno, že implementace funguje v identifikovaných okrajových případech, bylo přidáno 59 unit testů.

Klíčová slova

Kaitai Struct, generátor parserů, serializace, parsování, binární formáty

Překlad názvu

Rozšíření generátoru parserů Kaitai Struct o podporu serializace

Contents

1 Introduction ... 1
1.1 Motivation ... 1
1.2 Financial support ... 2
1.3 Goal ... 3
1.4 Outline ... 3
2 Binary format fundamentals ... 5
2.1 Basic concepts ... 5
2.2 Operating system calls ... 5
2.3 Structure of binary formats ... 6
2.4 Data types ... 6
2.4.1 Characters ... 6
2.4.2 Numbers ... 7
2.4.3 Options to delimit length of a field or array ... 7
3 Serialization principles ... 11
3.1 Text vs. binary serialization ... 12
3.1.1 Text serialization formats ... 12
3.1.2 Binary serialization formats ... 13
3.2 Application-specific vs. single-domain interchange formats ... 13
3.3 Schema-less vs. schema-driven serialization formats ... 14
4 Related work ... 17
4.1 Construct ... 17
4.2 BinData ... 18
4.3 Apache Daffodil ... 19
4.4 Comparison with Kaitai Struct ... 20
5 Kaitai Struct ... 21
5.1 Overview ... 21
5.2 Kaitai workflow ... 21
5.3 Compiler ... 22
5.4 Runtime libraries ... 22
5.5 KSY language ... 22
5.5.1 Features of the language ... 23
5.5.2 Data types ... 24
5.6 Format gallery ... 26

5.7 Web IDE ... 26
5.8 Kaitai Struct visualizer (ksv) ... 27
5.9 ksdump ... 28
5.10 Automated testing system ... 29
6 Analysis and design of serialization ... 35
6.1 Requirements ... 35
6.2 API of generated modules ... 36
6.2.1 Current state ... 36
6.2.2 Changes for serialization ... 37
6.3 General serialization procedure ... 38
6.4 Consistency checks: the _check() method ... 39
6.5 Implemented consistency checks ... 40
6.6 Analysis of KSY features ... 42
6.6.1 User-defined types ... 42
6.6.2 Fixed contents and validated fields ... 45
6.6.3 Value instances ... 46
6.6.4 Lengths and offsets ... 47
6.6.5 Parse instances ... 47
6.6.6 Parameters ... 48
6.6.7 Bit-sized integers ... 49
6.6.8 Fields delimited using "size", "terminator" and "pad-right" keys ... 49
6.6.9 Consistency checks that cannot be done in _check() ... 52
7 Serialization implementation for Java ... 53
7.1 Runtime library ... 53
7.2 Compiler ... 55
8 Testing ... 57
8.1 Testing areas ... 57
8.2 Testing write functionality ... 57
8.2.1 Method 1: compare serialized output to reference binary files ... 57
8.2.2 Method 2: write-read roundtrip ... 58
8.3 Testing consistency checks ... 60
8.4 Evaluation ... 61
9 Conclusion ... 63
9.1 Effects on Kaitai ecosystem ... 63
References ... 65
Appendix A: Source code ... 68

List of figures

Figure 1: A variable-length field "message" prefixed by a length field ... 8

Figure 2: A variable-length field "message" delimited by a terminator byte ... 8

Figure 3: Value 16 384 represented in VLQ format ... 9

Figure 4: Parsing and serialization ... 11

Figure 5: Application-specific vs. single-domain interchange formats ... 14

Figure 6: Schema-less and schema-driven data representation [8] ... 15

Figure 7: Two steps of Kaitai workflow for using a Python parser ... 22

Figure 8: GraphViz diagram of the .ico format generated from KSY specification ... 23

Figure 9: Kaitai Web IDE (with loaded PNG sample file + png.ksy) ... 27

Figure 10: Kaitai Struct visualizer ... 27

Figure 11: The process of testing parsers generated from KSY specifications ... 30

Figure 12: Kaitai Struct CI dashboard ... 31

Figure 13: _parent and _root references in a Kaitai Struct object tree ... 44

Figure 14: 3 passes of serialization ... 48

Figure 15: Integer field “len_message” specifying the size of a variable-length field “message” ... 50

Figure 16: Variable-length field delimited by a terminator byte ... 50

Figure 17: Anatomy of the reserved space of a field using “size”, “terminator” and “pad-right” (assuming “include: false”, i.e., the default setting) ... 51

Figure 18: Groups of methods of class KaitaiStream in the Java runtime library ... 55

Figure 19: 4 phases of write-read roundtrip ... 59

Figure 20: Hierarchy of Java test classes showing the implementation of write-read roundtrip ... 60

List of abbreviations

- API application programming interface
- ASCII American Standard Code for Information Interchange
- AST abstract syntax tree
- CI continuous integration
- EOF end of file
- GPLv3 GNU General Public License, version 3
- GUID Globally Unique Identifier
- IDE integrated development environment
- IEEE Institute of Electrical and Electronics Engineers
- KS Kaitai Struct
- KSC Kaitai Struct compiler
- KST Kaitai Struct Test
- KSY Kaitai Struct YAML
- MSb most significant bit
- NGI Next Generation Internet
- OOP object-oriented programming
- OS operating system
- UTF-8 Unicode Transformation Format (8-bit)
- UUID Universally Unique Identifier
- VLQ variable-length quantity
- YAML YAML Ain't Markup Language
- W3C World Wide Web Consortium

1 Introduction

Binary files are used to store and transfer data due to their simplicity and efficiency. Unlike text files, which are made up entirely of human-readable characters, binary files are general sequences of bits in one of two states, usually represented as 0 and 1. The analysis of digital data is at the core of many scientific disciplines such as reverse and forensic software engineering, cybersecurity, networking, and communication. To extract the required information from a file or data stream, a parser is used to parse (dissect) a sequence of bytes based on the structure of the binary data format. The output of the parser is information in a form that can be further manipulated.

Serialization is the opposite process – it converts data structures or objects into a binary format that can be stored or transmitted. It can be used for a variety of purposes. Serialized binary data can be sent over a network or between different systems, allowing interoperability; they can be used to store the state of an object and restore it later, allowing data persistence across different sessions or applications; they can be used to pass data between different processes or machines, facilitating communication in distributed systems.

Over the years, many tools have been developed to efficiently parse binary files and serialize data structures to them. The most used tools for working with binary formats include binary editors, binary analysis libraries, disassemblers and decompilers, packet sniffers and protocol analyzers, and custom analysis scripts.

Kaitai Struct (KS) is a free and open-source software project developed on GitHub. It is a tool for working with binary formats. Kaitai Struct introduces a declarative domain-specific language for describing the structure of arbitrary binary formats. A unique feature of Kaitai Struct is its programming language agnosticity. Format specification in Kaitai Struct is independent of the choice of programming language, so each specification can be automatically compiled into the 11 supported target languages. A compiler that does this can easily be extended to other target languages.

Since its launch in 2016, Kaitai Struct has become a respected tool for parsing binary formats. There are more than 650 projects on GitHub that use Kaitai Struct. It has found applications in reverse engineering/malware/security research, maintaining compatibility with legacy formats, digital preservation efforts, working with media files and transport protocols, satellite communications, scientific and university research (e.g., astrophysical and geophysical data processing), game development, unpacking and parsing files contained in firmware, etc.

1.1 Motivation

The current state of Kaitai Struct only allows you to extract data from binary files created by other programs (parsing). However, in many cases, the opposite direction is also needed, i.e., to modify the data in binary files or to create new ones (serialization). This has been by far the most requested feature in Kaitai Struct for a long time, and it is still missing. This prevents many users from using Kaitai Struct to its full potential.

Serialization is a logical extension to Kaitai Struct that immediately allows to use written format specifications not only for parsing, but also for serialization. The specifications that have been created are a result of many thousands of hours of work by over 80 contributors, and adding support for serialization will be a fraction of the time in comparison, while greatly expanding the use of all these specifications. As with parsing, the uses would be very diverse. In particular, the combination of parsing and serialization opens up a lot of room for additional uses and will attract many new users. Demands are mainly in the following areas:

- transport protocols (e.g. an easy way to communicate and exchange data between different programming languages in the same obscure protocol)
- converting files between different formats (e.g. multimedia)
- scientific purposes – e.g. a number of obscure formats are used in scientific measurements, often specific to a given manufacturer. The programs used to analyze the measured data are not familiar with these formats, so it is necessary to convert the original format to some other format that the program can work with (i.e., parse the obscure format first and then serialize it to a known format)
- easy fuzzing of hand-written parsers that parse a particular binary format – this is useful for revealing security vulnerabilities (example: a PNG file with all the checksums matching, but bogus content)
- repair, recovery and error correction of corrupted data on media (hard drives) or during network transmission
- video game modding (by editing obscure game formats)

I got involved in the development of Kaitai Struct in 2019. In November that year, I was invited to join the Kaitai development team. In May 2020, I accepted an offer from the founder Mikhail Yakshin to become the project administrator.

My main activity is development – I have created 2227 commits (as of 19.4.2024). I also review code and merge pull requests, submit ideas and visions in the form of issues, comment on issues and pull requests (2343 comments as of 19.4.2024), answer questions on Gitter, work on documentation, maintain the project and release new versions of the project.

1.2 Financial support

For my project "Serialization for Java and Python" in 2022 I received a grant[1] from the Dutch foundation NLnet, which provides financial support for open-source projects.

[1] https://nlnet.nl/project/Kaitai-Serialization

The project includes adding serialization support for Java and Python to the KS compiler and runtime libraries and writing online documentation. Everything is published in the Kaitai Struct project's GitHub repositories[2].

I successfully completed the project in the spring of 2023. This eased my way to receive another grant from the NLnet Foundation from the NGI Zero Entrust[3] fund in the summer of 2023.

1.3 Goal

The goal of this thesis is to add serialization support to the open-source tool Kaitai Struct, which includes design, implementation in Java and testing. Unlike the grant project, I will only discuss Java in the implementation section. I have chosen this narrowing down for better clarity of the work. The principles and conceptual design of serialization are the same for all languages, the implementation must consider the syntactic and behavioral differences and conventions of each target language, but it is more about the details.

1.4 Outline

Chapter 2 deals with the structure and typical features of binary formats.

Chapter 3 discusses the types and principles of serialization.

Chapter 4 gives an overview of related tools and projects.

Chapter 5 introduces Kaitai Struct, its characteristic features and its components.

Chapter 6 introduces the requirements on serialization support, analyzes KSY features and designs the API and serialization process in Kaitai Struct.

Chapter 7 describes the implementation of serialization in the Java runtime library and KS compiler.

Chapter 8 discusses test areas, methods, and evaluation.

[2] https://github.com/kaitai-io

[3] NGI0 Entrust is made possible with financial support from the European Commission's Next Generation Internet programme.

2 Binary format fundamentals

This chapter covers the basic design principles of binary formats. It describes common building blocks that can be found in almost all binary formats. The structures of specific binary formats vary, so you should refer to the relevant specifications to work with that format.

2.1 Basic concepts

Any computer file is a sequence of bytes with a certain length (also known as the file size). A byte is usually the smallest unit that a computer can address directly. It is a sequence of 8 bits. A bit is a binary digit, either 0 or 1. Therefore, the possible values of a byte are all bit patterns from 0000 0000 to 1111 1111. Bytes are often displayed in hex notation, which is done by encoding every 4 bits to a hexadecimal digit 0, …, 9, A, … F. This maps each byte to a two-digit hexadecimal number between 00 and FF. Software called hex editor or hex viewer can display the raw contents of any file in a hex dump, which is a listing of bytes formatted as 2-digit hex numbers.

A binary format is a set of rules that specify mapping between individual bytes and data attributes, which represent the meaningful information that the format is designed to store. The data attributes can have various data types, for example integer, enumeration, floating-point number, text string, and more. They are often grouped into objects (structures) and lists and form a structured data tree (also called object tree later in this text).

The process of converting a data tree to a byte sequence is called serialization, the conversion of a byte sequence to a data tree is called parsing (or deserialization). Since a byte sequence is how file contents are stored on a computer medium, parsing is often associated with reading and serialization is associated with writing.

2.2 Operating system calls

One of the tasks of the operating system is to provide an abstraction for working with files. It typically provides the following functions (possibly with minor variations):

- `read(num_bytes: uint): byte[num_bytes] | "EOF error"`
- `tell(): uint`
- `seek(offset: uint): void`
- `write(data: byte[]): void`

All these methods access the current byte position (offset) associated with the open file. This offset is common to both read and write operations. The `tell()` method returns this position. Usually, it is set to 0 when the file is opened, which points to the first byte of the file.

The `read()` method extracts the given number of bytes from the file starting at the current offset. If the file is long enough to satisfy this request, the bytes are returned and the current offset is incremented by the number of bytes read. Otherwise, an EOF (end of file) error is returned.

The `write()` method overwrites the file contents starting at the current offset with the given bytes and increments the offset by the number of bytes written. The file length is extended if necessary.

The `seek()` method repositions the current file offset to the given value. If the requested position is beyond the EOF, the file size will not change. Only when `write()` is called at this point, the gap between the last EOF position and the file offset is (at least conceptually) filled with null bytes `'\0'`, the bytes are written and the file size is updated [9].

2.3 Structure of binary formats

In most cases, binary formats are designed to be parsed sequentially from the beginning of the file (i.e., starting from the first byte). This can be done simply by successively calling `read()` with (possibly) different lengths depending on the size of the fields. In this case, the binary layout of format fields is simple – the first field starts at byte 0, and each subsequent field starts right after the end of the previous one (for example, if both the first and the second field are 4 bytes long, then the first occupies bytes 0-3 and the second 4-7).

However, the `seek()` method gives us one additional capability. We can jump to an arbitrary offset in the file and continue reading/writing there. The position where to seek is usually stored in the file directly as an integer that gives an offset relative to the beginning of file or some other known point. The advantage of this random access is that it is possible to retrieve specific information without having to read the entire file [10].

2.4 Data types

2.4.1 Characters

Since only binary patterns can be stored in digital files, graphical characters must be encoded into binary form before they can be written. This is governed by a character encoding, which defines the set of supported characters and the way to represent them in a binary form.

The most universally accepted standard is ASCII. It assigns meaning to each of the lower 128 values of a byte (i.e., from 00 to 7F in hex, or from 0 to 127 in decimal). This includes 95 printable characters: digits 0 to 9, uppercase letters A to Z, lowercase letters a to z, and punctuation symbols [1]. The rest are control characters.

ASCII is originally a 7-bit encoding. Therefore, it only defines meaning of the bit patterns of a byte with the most significant bit clear (i.e., in the format 0xxx xxxx). It does not use the upper 128 patterns of a byte, i.e., 1xxx xxxx. This allowed many 8-bit extensions of ASCII to be created, which leave the character-mapping of the lower 128 values of a byte intact (thus keeping backward compatibility with ASCII) and only define special meaning for the upper half of the byte range.

Nowadays, the dominant character encoding in most areas is UTF-8. It is backward compatible with ASCII. Unlike ASCII, which only supports a limited set of English alphanumeric characters and some special symbols, UTF-8 can encode any Unicode character (and Unicode covers most major writing systems in use today [2]). It is a variable-length encoding (it encodes code points in one to four bytes). The first byte of a sequence encoding a code point determines how many continuation bytes are part of the sequence [3].

Regardless of the character encoding used, the binary representation of a string is generally a concatenation of binary representations of individual characters.

2.4.2 Numbers

Integers in binary formats usually have a fixed length and therefore a fixed range. Typical sizes are 8, 16, 32 or 64 bits (which means 1, 2, 4 or 8 bytes). They can be unsigned or signed. Unsigned integer types can only represent non-negative numbers, whereas signed types also support negative. The most common method of representing signed integers is two's complement.

Fractional numbers are almost always represented in the 32-bit (single-precision, 4 bytes in size) or 64-bit (double-precision, 8 bytes in size) floating-point format according to the IEEE 754 standard. Some binary formats also use fixed-point arithmetic, which means the number is stored as an integer (signed or unsigned) that is to be multiplied by a fixed scaling factor [4].

2.4.3 Options to delimit length of a field or array

In binary formats, there are often fields with variable length. For example, a character string or an array of records. There are several common ways to delimit a variable-length field.

One way is to prefix the field with an unsigned integer that specifies the byte size of a field or the number of elements of an array that follows, see Figure 1. This has the advantage that there is no limitation on the value of the repeated unit (there is no special value that ends the repetition).

byte value (hex)	0b	48	65	6c	6c	6f	20	77	6f	72	6c	64
byte meaning	11	H	e	l	l	o	␣	w	o	r	l	d
field purpose	message length	message										

Figure 1: A variable-length field "message" prefixed by a length field

Another way is to use a terminator (see Figure 2). In the set of all possible values of the unit we are repeating (a byte, a multi-byte integer, or a structure), we reserve some subset to indicate that it is the last unit.

It is often used with text strings. It comes from the C language, where the standard representation of a string is just a pointer to the first character (char *), and the end of the string is found by sequentially scanning for the null character '\0'.

byte value (hex)	48	65	6c	6c	6f	20	77	6f	72	6c	64	00
byte meaning	H	e	l	l	o	␣	w	o	r	l	d	\0
field purpose	message											message terminator

Figure 2: A variable-length field "message" delimited by a terminator byte

Another example of a terminator-delimited field is a variable-length quantity (VLQ). It is a universal code that uses an arbitrary number of bytes to represent an arbitrarily large integer [5].

In the case of VLQ, the repeated unit is called a VLQ octet. It uses the most significant bit (MSb) to indicate whether another VLQ octet follows. The remaining 7 bits store a part of the integer value.

If the MSb is 0, then this is the last VLQ octet of the integer. If it is 1, then another VLQ octet follows [5]. An example is shown in Figure 3.

byte value (hex)	81		80		00	
byte value (bin)	1	000 0001	1	000 0000	0	000 0000
field purpose + value	is followed by more VLQ octets (1: **yes**)	bits 20..14 of the resulting integer	is followed by more VLQ octets (1: **yes**)	bits 13..7 of the resulting integer	is followed by more VLQ octets (0: **no**)	bits 6..0 of the resulting integer

Figure 3: Value 16 384 represented in VLQ format

The above example represents the value 16 384. We can check this by concatenating the integer bits from individual VLQ octets: this gives us 000 0001 | 000 0000 | 000 0000. The only bit set to 1 is at position 14, which has the weight 2**14 = 16384.

Another possibility to delimit a variable-length field is to reserve the rest of the stream for it. The stream may be either the entire file we are parsing or its substream. A substream is a portion of a larger stream which delimits the area for parsing certain structures. Usually, the length of a substream is specified by an integer field. The structures we are parsing inside the substream are confined in that substream; reading past the end of the substream is an error, even if the original stream continues after that point.

If the rest of the stream is reserved for a variable-length field, its size is the remaining number of bytes in the stream. If the rest of the stream is reserved for an array of records, we should repeatedly read records until there are no bytes left in the stream.

3 Serialization principles

This chapter defines the concept of serialization, the types of serialization, and the principles on which they are based.

Serialization is the process of translating data structures into a format that can be transmitted or stored. Deserialization is the process of reconstructing data structures from a serialized data representation [6].

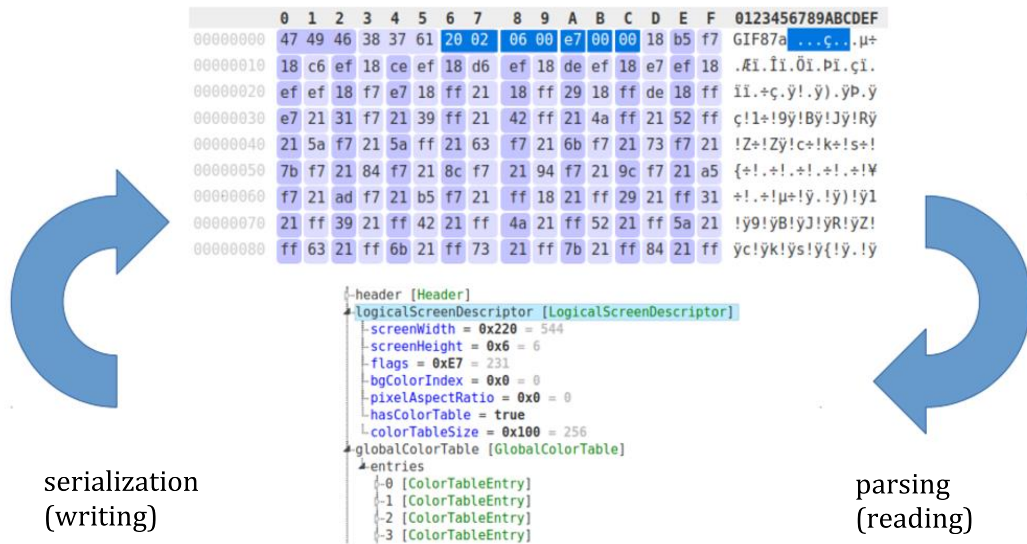


Figure 4: Parsing and serialization

The data structures to be serialized are usually the at-runtime representations of data in a software program, for example in the form of C-type structs or object states in object-oriented programming (OOP). Data structures (or objects) are in this context defined as instances of a data type, denoting that they are filled with concrete values [7].

Serialization can be classified based on many characteristics.

One distinguishing characteristic may be whether the serialized data is represented in text or binary form.

We can also classify serialization based on the range of intended consumers of the format, i.e., whether it is sufficient that the application writing the format can read it, or whether the serialized data must be readable by other existing applications.

Another important aspect is whether the serialized string can be deserialized without prior knowledge of its structure. If so, the serialization specification is schema-less; otherwise, the format is schema-driven [8].

3.1 Text vs. binary serialization

A serialization format is textual if its serialized form is a sequence of characters in a text encoding such as ASCII or UTF-8. The advantage is that there are a lot of computer tools available for manipulating text files, which makes textual formats perceived as human friendly [8]. They are easy to edit manually in a plain text editor.

In contrast, binary formats are not human-readable, but they are more efficient. They tend to use less memory, because they usually store numeric values in more compact formats (such as signed/unsigned integers or IEEE 754 floating-point numbers), rather than as text characters [10].

In addition, binary formats also offer advantages in terms of speed of access. While the basic unit of information is very straightforward in a plain text file (one byte equals one character), finding the actual data values is often much harder. For example, to find the third data value on the tenth row of a CSV file, the reader software must keep reading bytes until nine end-of-line characters have been found and then two delimiter characters have been found. This means that, with text files, it is usually necessary to read the entire file to find any particular value [10].

Binary formats can have some sort of map structure with offsets (pointers) of contained entries. The advantage of having such a map is that any entry within the file can be found by seeking directly at the recorded offset, without having to read the entire file [10].

3.1.1 Text serialization formats

Among the most common text-based serialization formats are the following [12]:

- CSV (Comma-Separated Values) – CSV is well suited to storing large amounts of tabulated data in a human-readable format [11]. It is a flat format – it does not support storing objects or structured data. There is no native support for other data types than strings – if the application needs to store other data types, such as integers/datetimes, it must handle the conversion from and to string itself.
- XML (Extensible Markup Language) – a markup language for storing and transmitting arbitrary structured data. It is well standardized, with plenty of tooling available to generate XML and validate it with schemas. One disadvantage of XML is its verbosity. The element name must be repeated in the end tag, which increases the overall size of XML data [11].
- JSON (JavaScript Object Notation) – JSON is a ubiquitous human-readable data serialization format that is supported by almost every programming language [11]. The core type is an object, which is an unordered collection of key-value pairs (where the key must be a string unique within the object, and the value can be any supported data type). Another important data type is an array, an ordered collection of arbitrary values. Unlike CSV or XML, where all values are strings, JSON defines a small set of basic data types such as number or boolean.

3.1.2 Binary serialization formats

Some examples of binary serialization formats are the following:

- BSON (Binary JSON) – A binary format for serialization of JSON-like documents. It deals with key-value pairs like JSON. Includes data types like datetime, byte array and others that are not part of the JSON spec [13]. Primary use is storage, not network communication [12].
- MessagePack – An efficient binary serialization format. It lets you exchange data among multiple languages like JSON [14]. Supports static typing. Better JSON compatibility than BSON. Primary use is network communication, not storage [12].
- Protobuf (Protocol Buffers) – A binary message format used to serialize structured data created by Google. Working with Protobuf involves defining the structure of data in a .proto file, from which generated code can be used to serialize and deserialize data with the specified structure.

3.2 Application-specific vs. single-domain interchange formats

When developing an application, there is often a requirement to serialize data to make it possible to store the application state/output on disk or transport the data over a network or to another application. Before deciding on what serialization format should be used, it is important to recognize whether in our use case the serialized output needs to be readable by applications we have no control over or not.

If not, all we need is an application-specific serialization format that we can choose arbitrarily. Arguably the simplest option is to use the native serialization approach offered by the programming language we are using – for example, the pickle module in Python or the "Serializable" interface in Java. The disadvantage is that the serialized object serialized using the native method in one language is not compatible with other languages. If this is a problem, we can use data interchange formats. A data interchange (also called exchange) format is domain-independent and can be used for data from any discipline [15]. Data interchange text formats include JSON and XML, for more efficiency we can choose from data interchange binary formats such as BSON, MessagePack, Protocol Buffers, Apache Avro.

An example when an application-specific format is suitable is when we have an application where the user works on some project (for example, editing a video or designing a printed circuit board) and we want to offer the user the ability to save the project to a file so that they can return to it later.

However, for the application to be useful, we often also want to offer export formats that are readable by other applications. For instance, when the user has finished editing a video, they would want to export it to a well-known video format so that they can show the video to people that do not have the editing software.

From the perspective of an application developer, it means that the format we are serializing into can no longer be arbitrary – it is in our interest to serialize to a format that is widely supported by the ecosystem of existing applications.

These well-known formats supported by a variety of applications are called single-domain data interchange formats. Single-domain means that such a format is only designed for one particular discipline. The term “data interchange” refers to the idea of allowing data to be shared between different computer programs [15].

See Figure 5 for a comparison between application-specific and single-domain interchange formats.

The concept of a single-domain data interchange format goes well together with the idea of open formats. An open format is defined by an openly published specification usually maintained by a standards organization. It can be used and implemented by anyone [16].

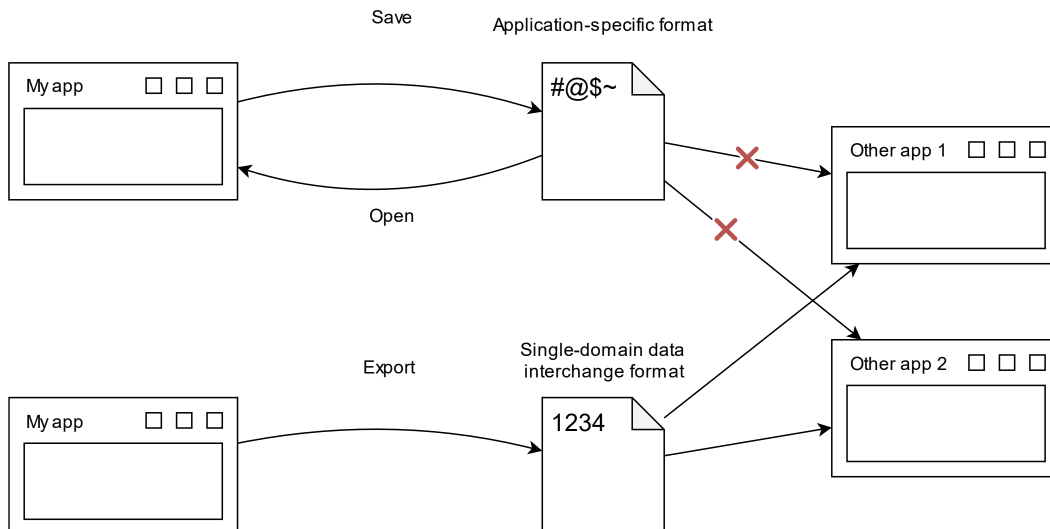


Figure 5: Application-specific vs. single-domain interchange formats

3.3 Schema-less vs. schema-driven serialization formats

A schema is a formal definition of a data structure. For example, a schema may describe a data structure as a sequence of two big-endian IEEE 754 single-precision floating-point numbers.

Schema-less serialization embeds the information provided by a schema into the resulting bit-strings to produce bit-strings that can be deserialized without additional information. This includes the names of the individual fields and their data types. In comparison to schema-driven serialization, schema-less serialization is perceived as easier to use because consumers can deserialize any bit-string produced by the implementation and not only those that consumers know about in advance [8].

In contrast, schema-driven serialization omits the structural information from the serialized output, making it more compact. To deserialize a string that has been serialized using schema-driven serialization, an external schema is required. Binary formats often use schema-driven serialization because it optimizes size and processing time.

An associative array (also known as a map) with two numeric properties, `latitude` and `longitude`, serialized with fictitious schema-less and schema-driven representations is shown in Figure 6. The schema-less representation (top) is self-descriptive and each property is self-delimited. In contrast, schema-driven representations (bottom) omit most self-descriptive information except for the length of the associative array as an integer prefix. A reader cannot understand how the schema-driven representation translates to the original data structure without a schema definition [8].

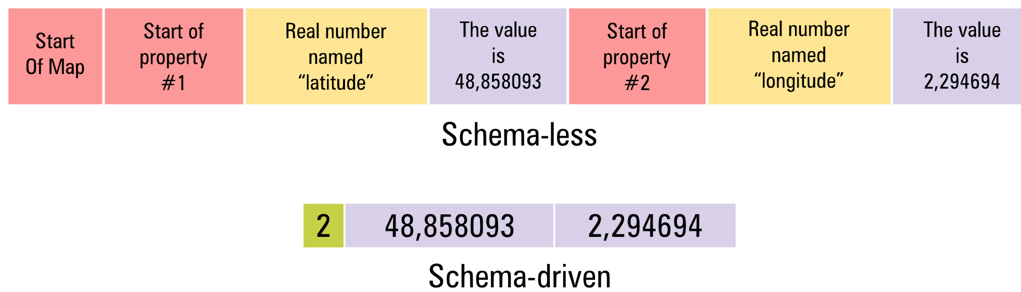


Figure 6: Schema-less and schema-driven data representation [8]

Both JSON and XML are schema-less formats. This is typical for text formats for data interchange in general. JSON implicitly includes the information about property names and data types, for example:

```
{
  "latitude": 48.858093,
  "longitude": 2.294694
}
```

We can infer several information about the shape of data just by looking at the serialized output. The overall type of the JSON data is "object" (an unordered collection of key-value pairs with unique string keys), which has properties `latitude` and `longitude`, both of type "number".

Binary data interchange formats can be schema-driven or schema-less. Schema-driven are for example ASN.1, Apache Avro, Microsoft Bond, Cap'n Proto, FlatBuffers, Protocol Buffers, and Apache Thrift. Schema-less binary serialization formats include BSON, CBOR, FlexBuffers, MessagePack, Smile, and UBJSON.

4 Related work

This chapter gives an overview of related tools, which deal with a similar problem as Kaitai Struct. Like Kaitai Struct, these tools are suitable when we need to serialize to a given arbitrary schema-driven binary format. We want to serialize data objects in memory populated with user-specified data to a specific binary format, which can be imported by other applications.

There are many tools that can only parse binary data (for example Spicy, Binary-parser in JavaScript, FlexT, EverParse), a small subset is capable of both parsing and serialization. I am not aware of any general-purpose tools that can only serialize. There are some tools focused on serialization for specific use, particularly for fuzzing – for example FormatFuzzer. This tool takes a binary template that describes the format of a binary input, for instance, a template for GIF. It generates an executable that produces test GIF data – also known as GIF fuzzer.

4.1 Construct

Construct is a Python library for parsing and serializing binary data.

Instead of writing imperative code to parse or serialize a piece of data, you declaratively define a data structure that describes your data. The structure is defined in Python code by combining building blocks defined by Construct, which are capable of serialization/deserialization [18]. As this data structure is not imperative code, you can use it in one direction to parse data into Python objects, and in the other direction, to build objects into binary data [19].

The library provides both simple, atomic constructs (such as integers of various sizes), as well as composite ones which allow you to form hierarchical and sequential structures of increasing complexity. Construct features bit and byte granularity, easy debugging and testing, an easy-to-extend subclass system, and lots of primitive constructs to make your work easier [19].

The most fundamental construct that can be serialized or deserialized is called a "field". There are many kinds of fields, each working with a different type of data (numeric, boolean, strings, etc.). Fields can be combined into more complex constructs – Structs and Sequences. A Struct is a collection of ordered and usually named fields parsed/built in that same order. When parsed, values are returned in a dictionary with keys according to the field names. A Sequence is like a Struct, but it returns the parsed values in a list rather than a dictionary [20].

Construct has been used to parse [19]:

- Networking formats like Ethernet, IP, ICMP, IGMP, TCP, UDP, DNS, DHCP
- Binary file formats like Bitmaps, PNG, GIF, EMF, WMF
- Executable binaries formats like ELF32, PE32

- Filesystem layouts like Ext2, Fat16, MBR

4.2 BinData

BinData is a Ruby library for reading and writing binary data.

The programmer specifies what the format of the binary data is, and BinData works out how to read and write data in this format. It is an easier (and more readable) alternative to Ruby's `#pack` and `#unpack` methods. Like Construct, it promises to replace imperative code that the programmer would have to write separately for parsing and serialization. It is enough to describe the binary data structure in a declarative way (still in Ruby code by combining BinData's facilities, like with the Construct library in Python), based on which BinData can parse and serialize the data of the given format.

The following example is from the BinData documentation [21]. It shows an imperative way to parse a binary structure using built-in Ruby methods like `#read` and `#unpack`. It is not very readable, and it is just code to parse the data: code to serialize the structure would have to be written separately, using the corresponding `#write` and `#pack` methods. This creates a lot of repetition, and the programmer is responsible for ensuring that the structure being parsed and serialized is the same.

```
io = File.open(...)
len = io.read(2).unpack("v")[0]
name = io.read(len)
width, height = io.read(8).unpack("VV")
puts "Rectangle #{name} is #{width} x #{height}"
```

In comparison, this is how the same structure can be parsed using BinData:

```
class Rectangle < BinData::Record
  endian :little
  uint16 :len
  string :name, read_length: :len
  uint32 :width
  uint32 :height
end

io = File.open(...)
r = Rectangle.read(io)
puts "Rectangle #{r.name} is #{r.width} x #{r.height}"
```

It is a bit longer, but more readable. We define a class `Rectangle` as a subclass of `BinData::Record`, which implements the `#read` and `#write` methods.

It supports all the common datatypes that are found in structured binary data. Support for dependent and variable length fields is built in [21].

4.3 Apache Daffodil

The Data Format Description Language (DFDL) is a language for describing both text and binary formats. DFDL builds on top of W3C XML Schema 1.0 and extends it with DFDL properties, which are defined as foreign attributes that are bound to the `http://www.ogf.org/dfdl/dfdl-1.0/` namespace. Although XML Schemas are traditionally only used to validate XML documents, the DFDL specification augments XML Schema to include serialization details so that it can accurately describe the contents of bit-strings [22].

The following snippet is an excerpt from the DFDL schema of the “shapefile” format [24]. Shapefile is a binary format for geographic information system (GIS) software [23].

You can see that the logical model for the data is described by XML Schema elements from the `http://www.w3.org/2001/XMLSchema` namespace, aliased as `xs` in the document. This defines that the Point type is a sequence of properties `X` and `Y`, both of which are floating-point numbers. Attributes from the `dfdl` namespace are used to specify that the length of both properties is 8 bytes and the byte order is little endian.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:dfdl="http://www.ogf.org/dfdl/dfdl-1.0/" ...>
  ...
  <xs:element name="Point">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="X" type="xs:double" dfdl:length="8"
            dfdl:lengthKind="explicit" dfdl:lengthUnits="bytes"
            dfdl:byteOrder="littleEndian" />
        <xs:element name="Y" type="xs:double" dfdl:length="8"
            dfdl:lengthKind="explicit" dfdl:lengthUnits="bytes"
            dfdl:byteOrder="littleEndian" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

Apache Daffodil is an open-source DFDL processor having both parser and unparser [25].

The main tasks that Daffodil can perform are parsing, unparsing, and generating a C parser/unparser (plus a few more). Parsing is the task of converting raw data into an abstract model called infoset, while unparsing is the reverse task of converting an infoset into the original format [26].

4.4 Comparison with Kaitai Struct

Construct and BinData use a different implementation approach than Kaitai Struct. They are both implemented as language-specific libraries: Construct is a Python library, BinData is a Ruby library. Their users use the provided interface directly from Python or Ruby code when creating a parser or serializer.

In contrast, Kaitai Struct works on the principle of generating source code from a language-agnostic YAML-based specification of the binary format structure. This design allows the use of a single specification from many programming languages, both to parse and serialize the described format.

Apache Daffodil supports both text and binary formats, whereas Kaitai Struct focuses only on binary formats. Like Kaitai Struct, it also uses a language-agnostic format to describe the binary format structure: an XML-based DFDL schema. However, Apache Daffodil is mainly a Java library. Users can interact with it via command-line interface or Java API. It is not primarily based on code generation like Kaitai Struct, but it can generate C code to parse or unparse data.

5 Kaitai Struct

This chapter presents the Kaitai Struct project and its components. The compiler generates parsers, which depend on the runtime library specific to the programming language. There are several visualization tools (Web IDE, ksv) which serve as user interfaces to Kaitai Struct. They make Kaitai Struct useful even to non-programmers, because they allow anyone to deeply inspect the internal data objects stored in binary files. The ksdump tool serves a similar purpose but provides the output in a machine-readable form.

5.1 Overview

Kaitai Struct is a universal parser generator for binary formats. It is free and open-source software developed on GitHub since 2016.

The main idea is to describe a particular format with the Kaitai Struct YAML (KSY) language. KSY is a declarative language used to describe various binary data structures, including binary file formats, network protocols, and more. The KSY format specification can then be compiled by a compiler into source code in one of the supported programming languages. These modules will contain the generated parser code that can read the described data structure from the file/stream and provide access to it in an understandable API. Another useful feature is the generation of diagrams in GraphViz format (see Figure 8).

5.2 Kaitai workflow

If you want to use Kaitai Struct from an application to parse data in a specific binary format, you need to perform 2 steps.

The first step is compilation. Take the Kaitai Struct specification, or format description, and compile it using the `kaitai-struct-compiler`. As output, you get the source code of the parser.

The main step is parsing. Give the binary file to the generated parser as input and you will get the parsed data as output. The Kaitai Struct parser works with the runtime library that you need to include in our application. See Figure 7 for an overview of both steps.

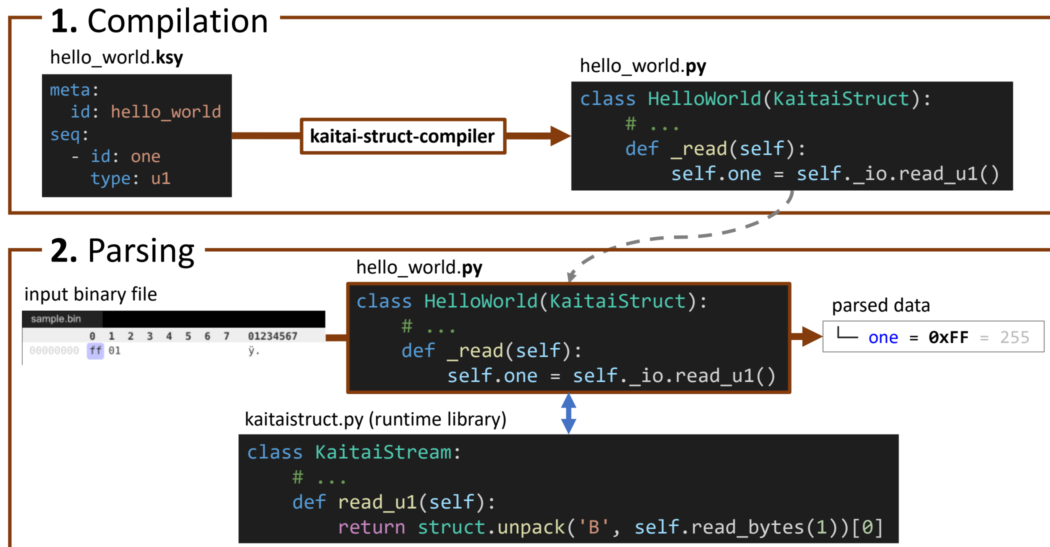


Figure 7: Two steps of Kaitai workflow for using a Python parser

5.3 Compiler

The core of the project is the Kaitai Struct compiler (KSC). It transpiles the format specifications in KSY into source code of the selected programming language. It is written in Scala. It currently supports 11 target programming languages: C++, C#, Go, Java, JavaScript, Lua, Nim, Perl, PHP, Python and Ruby.

5.4 Runtime libraries

Another part of the project is runtime libraries that make it easier to work with the byte stream. The code we receive from the compiler must be accompanied by a language-specific runtime library, which allows the generated code to be more concise and readable. All runtime libraries have the same set of elementary functions (called methods), which the compiler then refers to in the code – e.g. `read_u8le` reads an 8-byte unsigned integer from the byte stream, `read_bytes_full` reads a byte array up to the end of the stream, the `eof` function can be used to query whether the current byte position is at the end of the stream (thus getting a true/false value), etc.

5.5 KSY language

KSY is a declarative domain-specific language based on the generic YAML markup language. “Declarative” means that it only describes what a given binary structure looks like internally. This is different from the imperative (procedural) paradigm that is common to programming languages, where we describe a step-by-step process of what to do when parsing or serializing. Because the KSY language is built on top of YAML, it is easy to write your own tools to read KSY specifications.

The fact that the Kaitai Struct language is declarative makes it possible to automatically visualize any described format in a GraphViz diagram (see Figure 8).

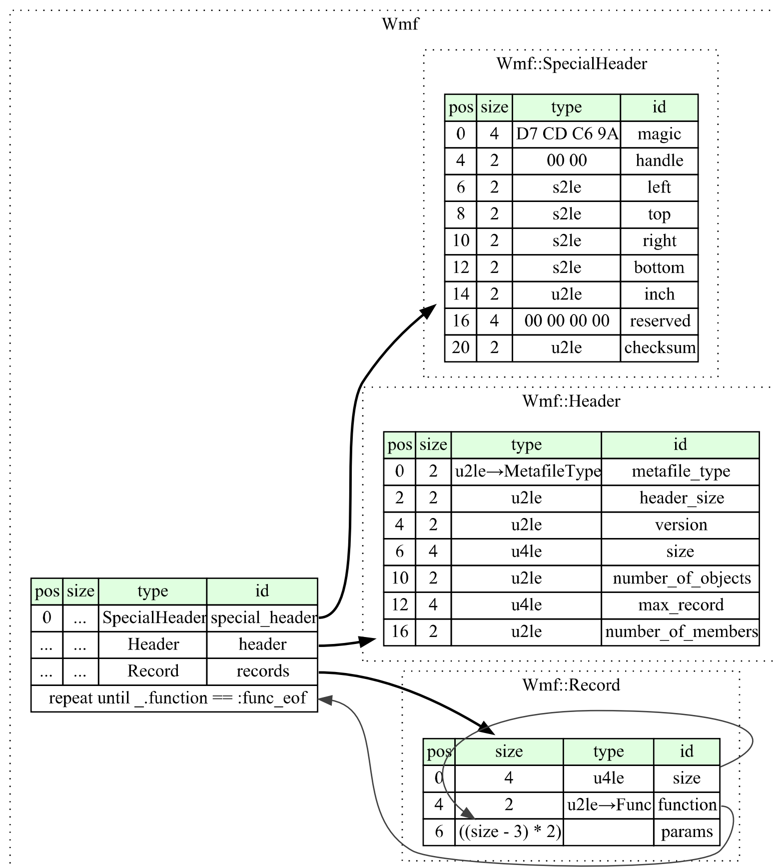


Figure 8: GraphViz diagram of the .ico format generated from KSY specification

5.5.1 Features of the language

The KSY language provides a set of features that cover various cases that occur in real-world binary formats. Since KSY is based on YAML, each feature is accessible via a specific YAML key (for example `seq`, `instances`, ...), after which a value of a certain type is expected.

- Sequential parsing (`seq`)
- Out-of-order parsing (`instances` with `pos`)
- Derived attributes (`instances` with `value`) – calculated from previously parsed attribute values
- Conditional parsing (`if`)
- Arithmetic and logical expressions – in `size`, `repeat-expr`, `if`, ... properties
  - `if: foo.bar == "T4"` (attribute is parsed only if equality holds)

    - `repeat-expr: (full_len - 4) / 6`

- Repetitions
    - to the end of the byte stream (`repeat: eos`)
    - a number of times known in advance (`repeat-expr: num_items`)
    - until the condition is valid (`repeat-until: _ == -1`)

5.5.2 Data types

You can use the `type` key in sequential fields (`seq`) and positional instances (`instances` with `pos`). An overview of possible values of the `type` key follows.

- **Built-in data types**
    - integers – `u1`, `s2`, `u8`, ...
    - floating point numbers (floats) – `f4`, `f8`
    - unaligned bit numbers and bitfields – `b1` (1 bit), `b2`, ..., `b64`
    - text strings – `str`
    - byte arrays – default type
    - enumeration types (defined in `enums`) – `enum: ...` in combination with an integer type
- **User-defined types** – composed of built-in types and other user-defined types

Let us walk through an example of a KSY specification adapted from the game/ftl_dat.ksy file in the format gallery [28].

```
meta:
  id: ftl_dat
  endian: le
seq:
  - id: num_files
    type: u4
  - id: files
    type: file
    repeat: expr
    repeat-expr: num_files
```

A .ksy specification starts with the `meta` section. The `id` key specifies the format identifier, which will be used as a basis for the name of the root class in the generated parser. The `endian` key sets the little-endian byte order as default.

The `seq` section is a sequence of attributes. The attribute name is in the `id` key. The type `u4` refers to an unsigned 4-byte integer. Note that Kaitai Struct uses byte widths

for numeric types with whole byte sizes, not bit widths like some other languages, which would call this type `uint32` (or similar).

The type `file` is a user-defined type, which will be defined in the next snippet. A field can also be repeated, so in this case the `files` attribute will be a list of elements of type `file`. The number of repetitions depends on the value of `num_files` that has been already parsed.

```
# ...
types:
  file:
    seq:
      - id: ofs_data
        type: u4
    instances:
      data:
        pos: ofs_data
        type: file_data
        if: ofs_data != 0
```

The above snippet shows the definition of the user-defined type `file`. User-defined type definitions belong to the `types` section.

We can see the already explained `seq` section with the `ofs_data` field. The `instances` section is another place where we can define fields. In this case, we have defined an attribute called `data` that starts at the byte offset determined by the value of `ofs_data`.

An attribute with an `if` key will be parsed/serialized only if the condition is true. The `if` key is one of many keys where you can use a powerful expression language included with Kaitai Struct.

```
types:
  # ...
  file_data:
    seq:
      - id: len_file
        type: u4
      - id: len_filename
        type: u4
      - id: filename
        size: len_filename
        type: str
        encoding: UTF-8
      - id: body
        size: len_file
```

The last part of the KSY example shows a use of the built-in string type (`type: str` used in the `filename` field). Using the `str` type requires specifying an explicit `encoding`. Another built-in type is a byte array, which is the implicit type for fields with known size that do not specify any `type` explicitly (such as `body` in this example).

5.6 Format gallery

The KSY language is used to describe different formats. The finished KSY format specifications can be found in the format gallery. As of 7 January 2024, it contains 185 format specifications. However, this number is by no means final, the format gallery has the most contributors and the fastest development. It includes, for example:

- image formats – BMP, EXIF, ICO, JPEG, PNG, DICOM (medical images), ...
- multimedia formats – AVI, MOV, WAV, OGG, MIDI, STL, ...
- archive formats – ZIP, RAR, gzip, LZH, phar, ...
- network protocols – DNS, ICMP, IPv4/IPv6, WebSocket, TLS, UDP, ...
- file systems – ISO9660 (CDs), FAT12, APM (disk partitioning table), VMWare virtual disk snapshots (.vmdk) and VirtualBox (.vdi), ...
- databases – .dbf (dBase), SQLite (.db, .sqlite), GNU gettext .mo (localization), ...
- executables – ELF (Unix systems), Java bytecode (.class), DOS MZ .exe, Microsoft PE (Portable Executable) .exe, Python bytecode (.pyc), ...

5.7 Web IDE

Kaitai Web IDE is a very useful tool for visualizing, developing, and debugging KSY format specifications. It is used to quickly preview the binary file and obtain the necessary information. Its user interface is shown in Figure 9.

The entire Web IDE runs in the browser, so no additional software needs to be installed. Any binary file can be loaded into it, which the Web IDE then parses based on the KSY specification. The result is structured data extracted from the input file, which is displayed in the object tree pane. The data inspector pane shows the parsed values starting from the current position in the byte stream, regardless of the KSY specification currently loaded. For most positions in the file, the values will not make sense. Another useful feature is to generate a parser in the selected output language.

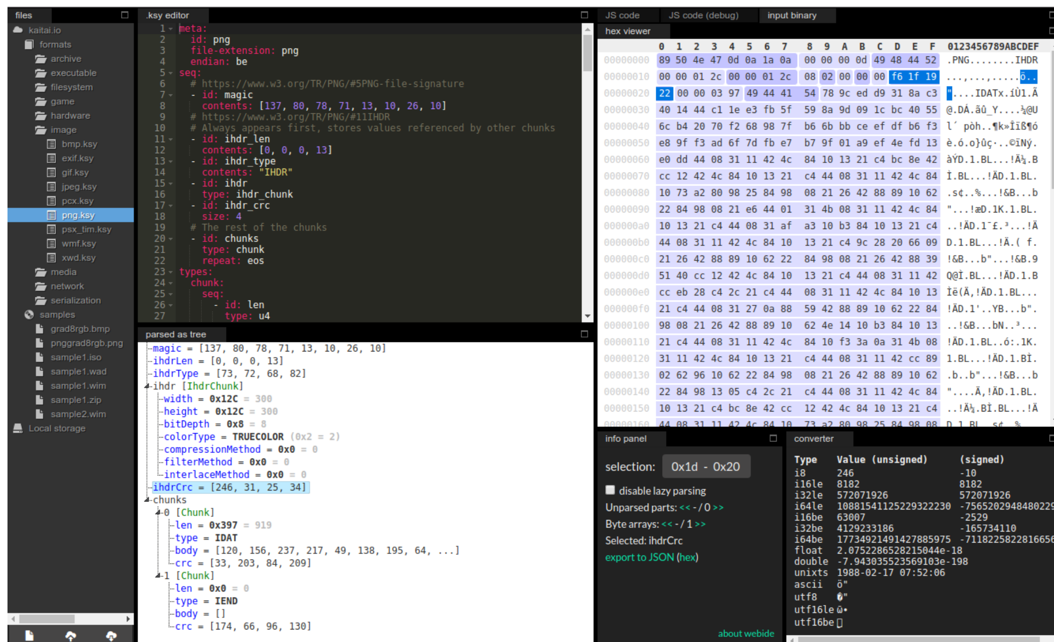


Figure 9: Kaitai Web IDE (with loaded PNG sample file + png.ksy)

5.8 Kaitai Struct visualizer (ksv)

The project includes the Kaitai Struct visualizer (ksv), an interactive console visualization tool (see Figure 10). It is an alternative to the Web IDE – it shows an object tree and a hex dump of the input file like the Web IDE.

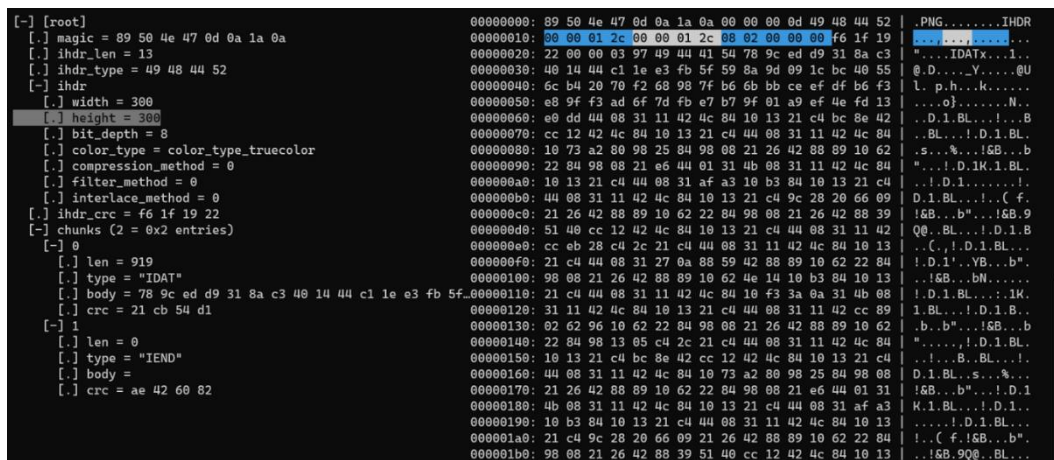


Figure 10: Kaitai Struct visualizer

It is written in Ruby, so it requires a Ruby installation. It is available in the RubyGems package repository, so it can be installed simply as `gem install kaitai-struct-visualizer` after installing Ruby. At runtime, it needs the Kaitai Struct compiler available on system PATH.

Once we have everything installed, we can run the visualizer from the command line as `ksv <binary-file> <ksy-file>`. The first argument is a binary file that we want to parse and the second argument is the .ksy specification that dictates the structure to be parsed.

You can navigate the object tree using the arrow keys (Up, Down, Left, Right) and the Home, End, Page Up, Page Down keys. You can exit the visualizer by pressing Q.

5.9 ksdump

The Kaitai Struct visualizer project includes two executables – `ksv` and `ksdump`. `ksv` is the interactive console visualizer discussed in the previous section. `ksdump` is a non-interactive command-line tool for dumping the same structured data that `ksv` displays to standard output. This is useful for automation. The usage is very similar to `ksv`: `ksdump <binary-file> <ksy-file>`. You can choose from YAML (default), JSON and XML output formats.

The following snippet shows the `ksdump`'s output in JSON format, showing the same data as the `ksv` screenshot (see Figure 10).


```
{
  "magic": "89 50 4E 47 0D 0A 1A 0A",
  "ihdr_len": 13,
  "ihdr_type": "49 48 44 52",
  "ihdr": {
    "width": 300,
    "height": 300,
    "bit_depth": 8,
    "color_type": "color_type_truecolor",
    "compression_method": 0,
    "filter_method": 0,
    "interlace_method": 0
  },
  "ihdr_crc": "F6 1F 19 22",
  "chunks": [
    {
      "len": 919,
      "type": "IDAT",
      "body": "78 9C ED D9 31 8A C3 40 14 44 C1 1E E3 FB 5F…",
      "crc": "21 CB 54 D1"
    },
    {
      "len": 0,
      "type": "IEND",
      "body": "",
      "crc": "AE 42 60 82"
    }
  ]
}
```

5.10 Automated testing system

Since Kaitai Struct must ensure that every feature is working in 11 programming languages, on different platforms, and on different compilers (especially in C++), it is necessary to periodically check that newly introduced changes to the compiler code or runtime libraries do not break the generated code in any of the target language/platform combinations (hereafter referred to as targets). This is very impractical to do manually precisely because of the number of targets that need to be checked.

Therefore, the project has an automated testing system (this is part of the so-called CI infrastructure) that is triggered whenever a new commit to the umbrella repository is pushed. It includes 233 test KSY specifications. See Figure 11 for an overview of the testing process. A complete parsing test consists of 3 parts: the KSY specification, a binary file and expected parsed values. Each KSY specification is compiled into parsers in all supported programming languages using the latest development version of the compiler. A binary file associated with the test is then processed by the parser and the parsing output is compared to the expected results. These are described in a simple Kaitai Struct Test (KST) language, whose specifications are used by the KST translator to generate unit tests with test assertions that verify that the equality of the expected and actual values of a variable holds. If all test assertions for a given test pass, the test is marked as „passed" for that target, otherwise it is marked as "failed".

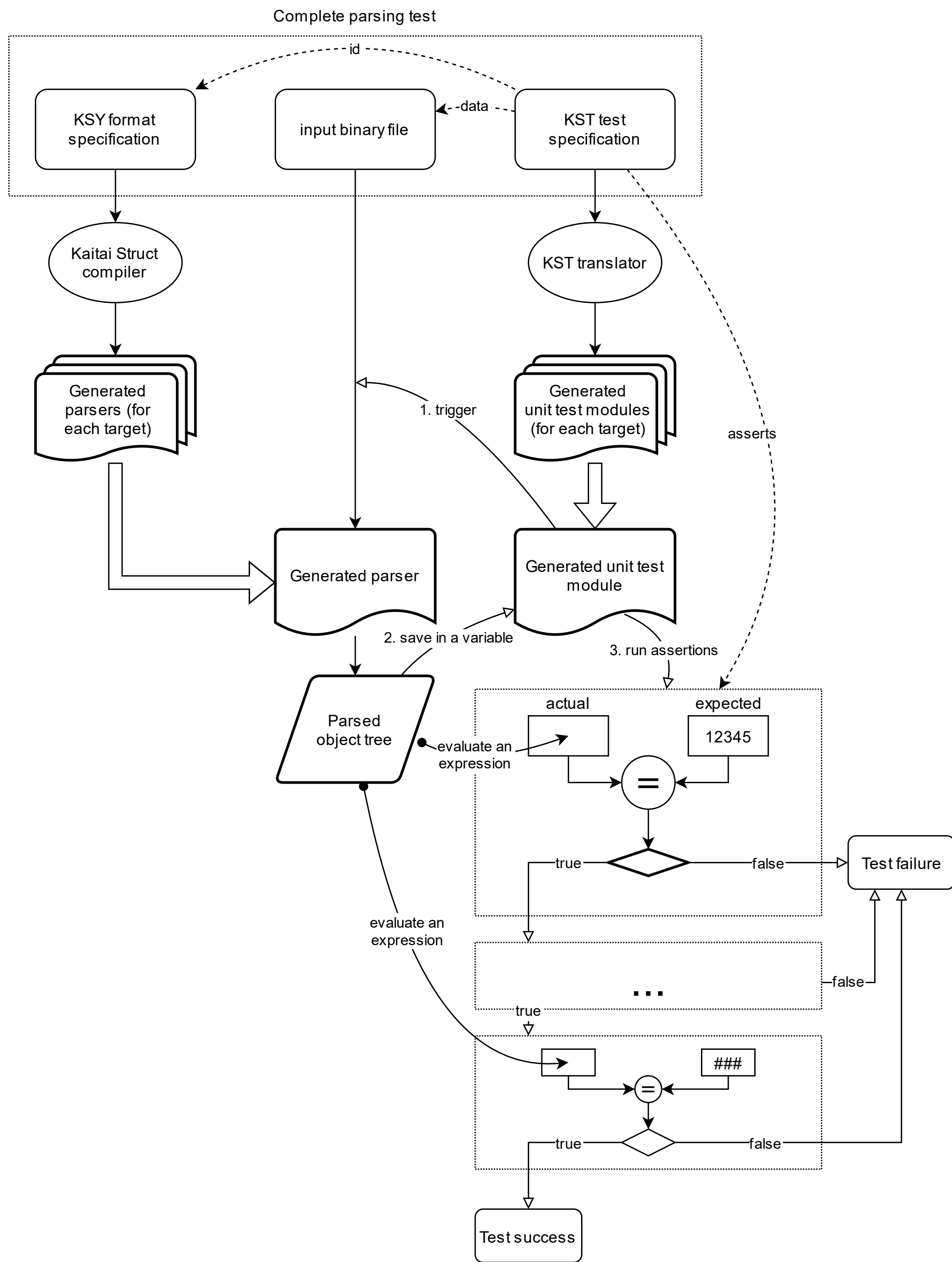
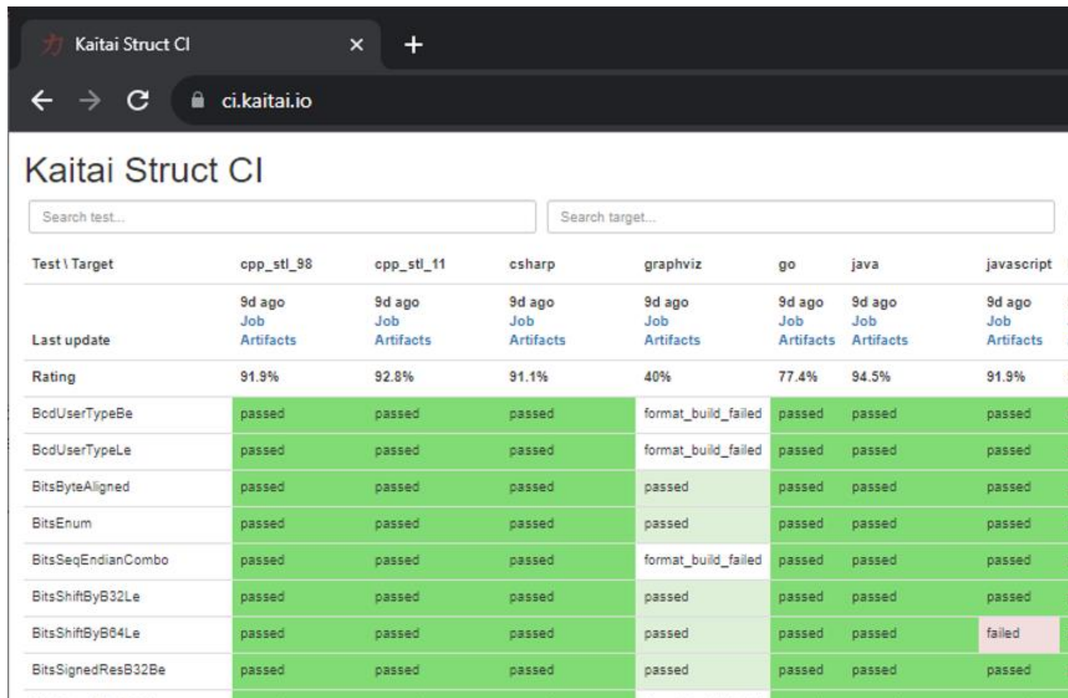


Figure 11: The process of testing parsers generated from KSY specifications

These results are then displayed in CI dashboard, which is shown in Figure 12.



Test \ Target	cpp_stl_98	cpp_stl_11	csharp	graphviz	go	java	javascript
Last update	9d ago Job Artifacts	9d ago Job Artifacts	9d ago Job Artifacts	9d ago Job Artifacts	9d ago Job Artifacts	9d ago Job Artifacts	9d ago Job Artifacts
Rating	91.9%	92.8%	91.1%	40%	77.4%	94.5%	91.9%
BcdUserTypeBe	passed	passed	passed	format_build_failed	passed	passed	passed
BcdUserTypeLe	passed	passed	passed	format_build_failed	passed	passed	passed
BitsByteAligned	passed	passed	passed	passed	passed	passed	passed
BitsEnum	passed	passed	passed	passed	passed	passed	passed
BitsSeqEndianCombo	passed	passed	passed	format_build_failed	passed	passed	passed
BitsShiftByB32Le	passed	passed	passed	passed	passed	passed	passed
BitsShiftByB64Le	passed	passed	passed	passed	passed	passed	failed
BitsSignedResB32Be	passed	passed	passed	passed	passed	passed	passed

Figure 12: Kaitai Struct CI dashboard

In Figure 12 you can see a small part of our CI dashboard. The target languages are listed in the columns and the individual tests in the rows. The cells of the table show the test status in each language – “passed” or “failed”.

Let us look at an example of an actual test called `ExprIntDiv`. It tests whether the integer division operation in the KSY expression language has the expected behavior, which is floor division (rounding the mathematical result of the division towards negative infinity, like the `//` operator in Python): the result of `-5 / 4` should be `-2`, provided that `-5` and `4` are integers.

The test KSY specification (expr_int_div.ksy) looks like this:

```
# Tests division operation, both positive and negative
# See https://github.com/kaitai-io/kaitai_struct/issues/746
#  => the KS division operation `a / b` should do `floor(a / b)`
meta:
  id: expr_int_div
  endian: le
seq:
  - id: int_u
    type: u4
  - id: int_s
    type: s4
instances:
  div_pos_const:
    value: 9837 / 13
  div_neg_const:
    value: -9837 / 13
  div_pos_seq:
    value: int_u / 13
  div_neg_seq:
    value: int_s / 13
```

Once we have the KSY specification, we can generate a parser using Kaitai Struct compiler and use it to parse arbitrary binary data, which will yield some results. However, for a test to be effective, we need to select one binary input and write a corresponding KST spec with the test assertions for expected values that match the desired behavior. The contents of the expr_int_div.kst file follow:

```yaml
id: expr_int_div
data: fixed_struct.bin
asserts:
  - actual: int_u
    expected: 1262698832
  - actual: int_s
    expected: -52947
  - actual: div_pos_const
    expected: 756
  - actual: div_neg_const
    expected: -757
  - actual: div_pos_seq
    expected: 97130679
  - actual: div_neg_seq
    expected: -4073
```

It specifies that the fixed_struct.bin file will be used as parser input. A list of asserts follows, each with `actual` and `expected` keys. Both are KSY language expressions.

This spec is consumed by KST translator (part of the kaitai_struct_tests repository), which generates unit test modules. For example, this is the generated Python test module for `ExprIntDiv`:

```python
# Autogenerated from KST: please remove this line if doing any
edits by hand!

import unittest

from expr_int_div import ExprIntDiv

class TestExprIntDiv(unittest.TestCase):
    def test_expr_int_div(self):
        with ExprIntDiv.from_file('src/fixed_struct.bin') as r:

            self.assertEqual(r.int_u, 1262698832)
            self.assertEqual(r.int_s, -52947)
            self.assertEqual(r.div_pos_const, 756)
            self.assertEqual(r.div_neg_const, -757)
            self.assertEqual(r.div_pos_seq, 97130679)
            self.assertEqual(r.div_neg_seq, -4073)
```

Test modules are responsible for calling the format class (generated by the compiler from the .ksy specification) on a specific binary file and then running a few assertions to check that the parsing works correctly.

6 Analysis and design of serialization

This chapter introduces the requirements on serialization support, analyzes KSY features and designs the API and serialization process in Kaitai Struct. This design will be used in all languages, but this thesis only discusses the implementation for Java.

6.1 Requirements

A basic requirement is to support two main use cases of serialization: editing an existing file and creating a new file from scratch. Editing means that the same objects holding the data parsed from a stream can be mutated and then serialized back to the same stream or to another stream – there is no need to recreate the objects. However, constructing new objects by setting each field should also be possible.

An essential requirement is that serialization implementation must be consistent with parsing (this comes from the fact that serialization is the inverse operation to parsing). This means that if we serialize an object tree and parse the serialized stream, we should get the original object tree again. The opposite is impossible in the general case (we should not expect to always get the original byte stream if we parse it to an object tree and serialize it), because parsing is not guaranteed to preserve information about every single byte contained in the input byte stream.

Consistency cannot be achieved if the object tree itself is inconsistent with the definition of the given format. That is, if such an object tree could never be a result of successful parsing, because it violates some property that all successfully parsed trees have. This makes it obvious that if we serialize this tree and parse the serialized output, we will not get that object tree back, regardless of whether serialization works or not. A user can create inconsistent objects by mistake, whose serialization will lead to corrupted files. To prevent this, one of the goals of Kaitai serialization is to provide a set of checks that detect when some consistency property is not satisfied.

An important goal is to allow reusing the existing KSY specifications without having to change them. One of the advantages of Kaitai Struct is that hundreds of formats have already been described in KSY specifications, so this goal enables the widest possible use of both the serialization support and, by extension, the specifications themselves.

Related to the previous goal, serialization should support all features, constructs and their combinations that are part of the KS language. If serialization did not work with some combination of constructs, users would have to adapt the specifications to work around the unsupported features so that the specification can be used for serialization. This would raise the barrier to being able to use existing specifications for serialization, making it less useful.

The primary requirement is generality: to cover all conceivable format specifications. A secondary goal is to work as autonomously as possible and minimize the need for

the user to interact with the serialization process, such as providing additional information or triggering sub-actions. These requirements are sometimes contradictory.

6.2 API of generated modules

Let us start by looking at what the current interface of generated modules looks like and how it needs to be changed to accommodate serialization.

6.2.1 Current state

Currently, one KSY specification is translated to one Java module with a single top-level class. The name of this class depends on the value of `/meta/id` in the specification – for example, if the `/meta/id` is `hello_world`, the class will be called `HelloWorld`. Its constructor has the following signature:

- `public HelloWorld(KaitaiStream _io, KaitaiStruct _parent = null, HelloWorld _root = this)`

The constructor has a required `_io` parameter of type `KaitaiStream`. This is a class defined in the Kaitai Struct runtime library for Java – it provides an abstraction over an I/O byte stream. Generated format classes always read from KaitaiStream objects.

Parameters `_parent` and `_root` are optional (they have default values). They allow navigation in the object tree: `_parent` refers to the parent node in the tree (it is `null` in the root node), `_root` refers to the root object of the tree. They can be used in user-specified expressions in KSY specifications.

The `HelloWorld` class has the following methods:

- `private void _read()`

The `_read()` method is responsible for parsing the `seq` structure. It is private because the user does not need to call it – it is called automatically in the constructor. This mode is called `autoRead` and it is enabled by default. We can disable it by passing `--no-auto-read` to the compiler, which makes `_read()` public and it will no longer be called in the constructor.

We can see that Kaitai Struct uses an underscore prefix `_` for methods and properties with special meaning that are added automatically (like `_read`, `_parent` and `_root`). This is to distinguish them from user properties that share the same namespace, because they are also translated into methods of the format class (but a user property specified in a KSY specification cannot start with an underscore, which avoids potential name collisions).

Then there are getters for each property accessible in the KSY specification. This includes user properties (`seq` fields, `instances` and parameters) and automatic properties: `_io`, `_parent` and `_root`. Despite Java convention, the getters do not use the `get` prefix.

- `public {TYPE} {userProperty}()`

- `public KaitaiStream _io()`
- `public KaitaiStruct _parent()`
- `public HelloWorld _root()`

For `seq` fields and parameters, the getter has a trivial implementation, for example:

- `public int one() { return one; }`

For parse and value instances, the getter parses/evaluates the instance and caches the value on the first call. Subsequent calls just return the cached value.

6.2.2 Changes for serialization

By default, serialization support will be disabled and read-only classes will continue to be generated as before. A new compiler command-line option `--read-write` will be added, which enables read-write mode. This adds the methods needed for serialization to the generated classes [29].

First, read-write mode disables the `autoRead` mode. The reason is that whereas in read-only mode the only action you can do is parsing, in read-write mode, it is no longer clear to Kaitai Struct why you are creating a particular object. The purpose may be just to create an empty object to be filled with data and later written, in which case you do not want to read from any stream. For this reason, `_read()` is never called automatically from class constructors in read-write mode – you need to call it explicitly if you want to read from a stream [29].

Setters for `_parent`, `_root` and user properties will be added. They use the `set` prefix: `set_parent()`, `set_root()` and `set{UserProperty}()`.

Furthermore, read-write mode adds the following methods:

- `public void _write(KaitaiStream io = this._io)`
- `public void _check()`

The `_write()` method serializes the object and child objects to the specified stream. If no stream is specified as an argument, the existing stream given in the constructor is used. The `_check()` method performs consistency checks; it should be called by the user before `_write()` to ensure the values satisfy the consistency properties (if they did not, the output would be corrupted). If some consistency check fails, a `ConsistencyError` is thrown.

Since `_read()` will not be called in the constructor in read-write mode, `_io` is no longer required to be specified to the constructor. It can be provided as an argument to the `_write()` method. Therefore, the possibility to create the format class with no arguments is added:

- `public HelloWorld(KaitaiStream _io = null, KaitaiStruct _parent = null, HelloWorld _root = this)`

6.3 General serialization procedure

Let us start with a simple example to see what the serialization API looks like. First, we compile the following .ksy specification in read-write mode:

```
meta:
  id: hello_world
  endian: le
seq:
  - id: foo
    type: s4
    repeat: expr
    repeat-expr: 2
```

This will generate a HelloWorld.java source file with class `HelloWorld`. We want to set `foo` to `[-4, 65536]` and write the structure to bytes. This is how we do it in Java code [29]:

1. Create a KS object	`HelloWorld hw = new HelloWorld();`
2. Set the object fields	`hw.setFoo(new ArrayList<>(Arrays.asList(-4, 65536)));`
3. Call _check() on each KS object	`hw._check();`

4. Call _write() on top-level object	`byte[] output = new byte[8];`
	`try (KaitaiStream io = new ByteBufferKaitaiStream(output)) {`
	`    hw._write(io);`
	`}`
	`// output: [fc ff ff ff 00 00 01 00]`

Note that there are essentially 4 phases of serialization in Kaitai Struct [29]:

1. Initialize an object instance of a KS-generated class (which reflects a user-defined type in the source .ksy specification).
2. Set the object properties (`seq` fields or positional instances in the .ksy) according to the data you want to serialize.
3. Call the `_check()` method of the KS object after setting its properties once it is ready for serialization.
4. Call the `_write()` method on the top-level object and pass the `KaitaiStream` object you want to write to.

First, we create an empty instance of the top-level class `HelloWorld` and bind it to the `hw` variable. As you can see in the original .ksy spec, it has only one field called `foo`, which is a list of two `s4` (signed 4-byte) integers. We assign such list with the values we wanted to write to the `foo` field using the `setFoo` setter in Java. After that, we believe that the `hw` object is ready to be written, so we call `hw._check()`. When it passes

(that is, it doesn't throw any exceptions, see section 6.4), we move on to the actual writing [29].

We prepare a byte array for the output, create a `ByteBufferKaitaiStream` as a wrapper around this byte array and then call the `_write()` method on the top-level `hw` object, which serializes it into the provided stream. After the try-with-resources statement, `output` holds the final byte data that we can, for example, write to a file or transfer over the network [29].

6.4 Consistency checks: the _check() method

Let us focus on what the `_check()` method does. We know that `foo` is expected to be a list of exactly 2 integers (because of `repeat-expr: 2` in the source .ksy). Every parsing of the `hello_world` type tries to read 2 integers, and in any successfully parsed `HelloWorld` object, `foo` will be always 2 elements long. See the generated `_read()` method for reference:

```
public void _read() {
    this.foo = new ArrayList<Integer>();
    for (int i = 0; i < 2; i++) {
        this.foo.add(this._io.readS4le());
    }
}
```

However, the `setFoo()` setter allows us to set any integer list – even if its length is 0, 1 or greater than 2.

```
public void setFoo(ArrayList<Integer> _v) { foo = _v; }
```

Nevertheless, if we set `foo` to a list of length other than 2 and write the `hw` object to bytes, we will not be able to get the same state of the `HelloWorld` object by parsing these bytes: either the parsing fails with an EOF exception if the stream was shorter than 8 bytes, or we get garbage values in `foo` (if the written `foo` had less than 2 elements, but the stream is long enough) because we interpret some bytes outside `foo` as if they were `foo` values, or we may read 2 correct values, but the object we serialized had actually more. In such cases, it is inevitable that not only the parsed `foo` would not match the `foo` we wrote, but also the offsets of all the fields after `foo` would be shifted, so their values would also be incorrect [29].

This is because by setting `foo` to anything other than a 2-integer list and serializing it, we violate the property of consistency – the data is not consistent with the constraints directly following from how the format is specified in the source .ksy file. Kaitai Struct knows these constraints, and generates assertions for them in the `_check()` method whenever possible. If `_check()` detects a consistency issue, it throws a `Consis-`

`tencyError`, telling you to fix the problem and try again. This protects you from proceeding to the writing phase with inconsistent values, which would inevitably result into corrupted data that cannot be faithfully decoded back to the original values [29].

6.5 Implemented consistency checks

Consistency checks are necessary for various reasons. This section provides an overview of the constraints that users must uphold when populating the KS objects with data so that the resulting objects are consistent. If any of these constraints are violated, the corresponding `_check()` method will throw a `ConsistencyError`.

Essentially all methods of delimiting the size of a field require some kind of consistency check. For example, if a field uses the `size` key that provides an upper bound on the length of the field's byte contents, under no circumstances can we fit byte contents longer than `size` into it.

Repetitions are in principle analogous to size delimiting keys, so their consistency checks are unsurprisingly similar (notice the similarity of `repeat: expr` to `size`, `repeat: eos` to `size-eos: true`, and `repeat: until` to `terminator` with `include: true`).

Finally, values of parameters must match the values with which a parent object would create a child object when parsing.

- **Size delimiting keys** [30]
  - `size`, no `terminator`, no `pad-right` – byte array length must be equal to `size`
  - `size` in combination with `terminator` or `pad-right` – byte array length must be less than or equal to `size`
  - `terminator`, `include: false` (default), optionally with `size` and/or `pad-right` – byte array must not contain the `terminator` byte
  - `terminator`, `include: true`, no `size`
    - `eos-error: true` (default) – the only `terminator` byte must be at the end of the byte array
    - `eos-error: false` – the `terminator` may be at the end of the byte array, but nowhere else; if it is not present, we must be at EOF after writing the field
  - `size` and `terminator`, `include: true`, no `pad-right` or `pad-right == terminator`
    - if byte array is shorter than `size` – the only `terminator` byte must be at the end of the byte array
    - if byte array length is exactly `size` – the `terminator` byte may (or may not) be at the end of the byte array, but nowhere else

  - `size` and `pad-right`, no `terminator` – the last byte (if any) must be different from `pad-right`
  - `size`, `terminator`, `include: false` (default) and `pad-right != terminator`
    - if byte array length is exactly `size` – the last byte (if any) must be different from `pad-right`
  - `size`, `terminator`, `include: true` and `pad-right != terminator`
    - if byte array does not contain `terminator` – the last byte (if any) must be different from `pad-right`
    - if byte array contains `terminator`, then it must be only at the end of the byte array
  - `size-eos: true` – same checks as for `size` + we must be at EOF after writing the field
- **Repetitions** [30]
  - `repeat: expr`
    - the length of the list must be equal to `repeat-expr`
  - `repeat: eos`
    - we must not be at EOF before any element
    - we must be at EOF after all elements
  - `repeat: until`
    - the list must not be empty (must contain at least one element, because there is always the element for which `repeat-until` is true)
    - the `repeat-until` expression must evaluate to true only for the last element and no other
- **Parameter values of user-defined types' child objects** [30]
  - `_root`: `child._root` must refer to the identical object as `_root` in the current object
  - `_parent` (assuming no parent overriding takes place – https://doc.kaitai.io/user_guide.html#_enforcing_parent_type) – `child._parent` must refer to the identical object as `this`, i.e., the current object
  - user-defined parameters (`params`) except I/O stream parameters – each parameter must match the value to which the expression written in the .ksy specification for the argument value evaluates
    - for primitive types and arrays – compare by equality
    - for objects of user-defined type – compare by identity

6.6 Analysis of KSY features

KSY specifications use certain fundamental constructs – built-in data types with many configuration options, user-defined types, sequences, parse and value instances, parameters, expressions. It is one thing to implement serialization for these core features individually. But they often occur in various combinations, which must be considered when designing the implementation of individual features, which presents a great challenge and often increases the complexity of the design.

6.6.1 User-defined types

Real-world .ksy specifications often define custom types in the `types` section. For example:

```
meta:
  id: user_types
  endian: le
seq:
  - id: one
    type: chunk
types:
  chunk:
    seq:
      - id: len_body
        type: u4
      - id: body
        size: len_body
```

A typical way to serialize such format would be as follows [29]:

```java
UserTypes ut = new UserTypes();

UserTypes.Chunk one = new UserTypes.Chunk(null, ut, ut._root());
one.setLenBody(2);
one.setBody(new byte[] { 'h', 'i' });
one._check();

ut.setOne(one);
ut._check();

byte[] output = new byte[6];
try (KaitaiStream io = new ByteBufferKaitaiStream(output)) {
    ut._write(io);
}
// output: [02 00 00 00 68 69]
```

First, we instantiate the root class `UserTypes` as usual. Then we need the instance of the user-defined chunk type, translated as `UserTypes.Chunk` in Java. We use the usual way to create an instance of a class, but this time using all 3 arguments of the constructor:

```java
public Chunk(KaitaiStream _io, UserTypes _parent, UserTypes _root)
{
    // ...
}
```

The reason is that we must provide values for the `_parent` and `_root` parameters. These built-in references should be valid in all KS types so that it is possible to rely on them in expressions inside the .ksy spec when needed. When you instantiate inner objects (any object instances of user-defined types other than the root object) manually, you have to set these properties correctly [29].

See Figure 13 for an overview of what the `_parent` and `_root` references should look like in an object tree. The `_root` property of all objects should refer to the root object, including `_root` of the root object itself. The `_parent` property refers to the parent node in the tree. The root object has no parent node, so its `_parent` should be set to `null`.

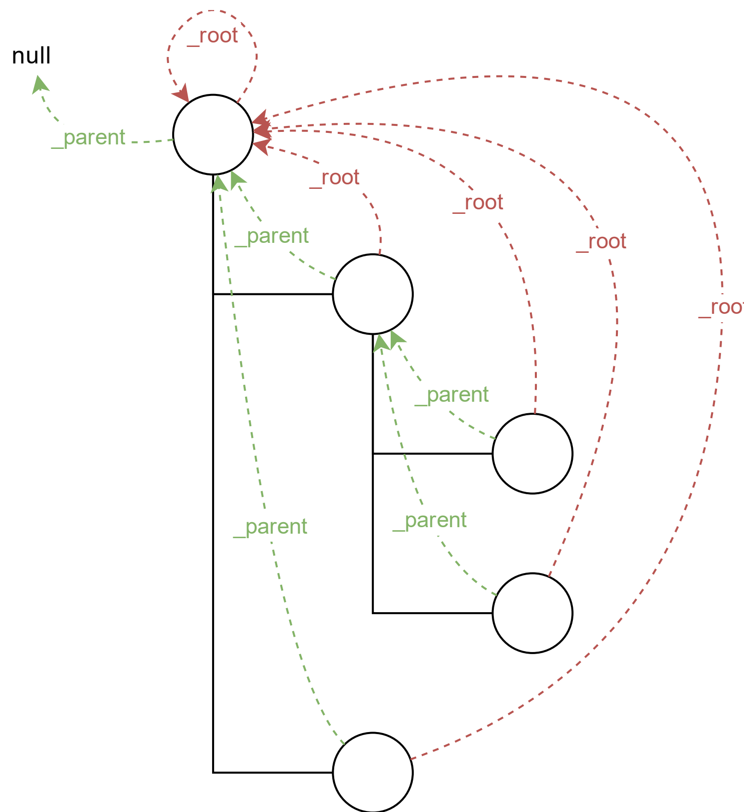


Figure 13: _parent and _root references in a Kaitai Struct object tree

If you do not set the correct values to both `_parent` and `_root`, it is a consistency issue that will be reported in `_check` of the parent object (`ut` in this case):

```
UserTypes ut = new UserTypes();

UserTypes.Chunk one = new UserTypes.Chunk(null, ut); // WRONG: we
did not pass "ut._root()" to "_root"!
one.setLenBody(2);
one.setBody(new byte[] { 'h', 'i' });
one._check();

ut.setOne(one);
ut._check(); // io.kaitai.struct.ConsistencyError: Check failed:
one, expected: org.example.UserTypes@539645a2, actual: null
```

The error message is a bit inconcrete at the moment, because it only says there is a problem with the field one but does not specify what exactly it is. This will be improved in the future, but for now, check out the line where the `ConsistencyError` was thrown for more details:

```
io.kaitai.struct.ConsistencyError: Check failed: one, expected:
org.example.UserTypes@539645a2, actual: null
    at org.example.UserTypes._check (UserTypes.java:48)
    ...
```

```
public class UserTypes extends KaitaiStruct.ReadWrite {
    // ...
    public void _check() {
        if (!Objects.equals(one()._root(), _root()))
            throw new ConsistencyError("one", one()._root(),
_root());
        // ...
    }
```

By looking into the generated code, we figure out that the `_root` parameter of field `one` had a wrong value. It should have been equal to `ut._root`, but it was `null`.

After we create an instance of the `UserTypes.Chunk` subtype, we set its properties, and then we call `_check`. This is important: `_check` always works only for the one object on which you call it, it does not recursively descend into substructures (unlike `_read` and `_write` which do that, so you call them just on the top-level object). So it is not enough to call `_check` just on the top-level object – you have do it for every KS object on which you use setters [29].

6.6.2 Fixed contents and validated fields

After creating a new KS object, you must also set fields with `contents` or `valid` on them, even if there is only one valid value they can have. Kaitai Struct does not set them automatically at the moment. For example, the following `magic` field:

```
meta:
  id: elf
  # ...
seq:
  - id: magic
    contents: [0x7f, "ELF"]
```

needs to be set as follows:

```
Elf e = new Elf();

e.setMagic(new byte[] { 0x7f, 'E', 'L', 'F' });
// ...
e._check();
```

The `_check` method validates such fields, so you get notified if the values are not valid.

6.6.3 Value instances

They do not have setters. If you need to make value instances change, you must set their inputs (fields they depend on). For example [29]:

```
meta:
  id: value_instances
seq:
  - id: len_data_raw
    type: u1
  - id: data
    size: len_data
instances:
  len_data:
    value: len_data_raw - 3
```

```
ValueInstances r = new ValueInstances();

r.setData(new byte[] { 1, 2, 3, 4, 5 });
r.setLenDataRaw(8);
System.out.println(r.lenData()); // => 5
```

We set a 5-byte array to `data`, so for the object to be consistent, we need `len_data` to be 5. Since it is defined as `len_data_raw - 3`, we set `len_data_raw` to 8, which makes `len_data` to be 8 – 3 = 5.

What happens if you want to change the length of `data` in this existing object? Value instances in KS are cached, so even if you change `len_data_raw`, `len_data` will keep returning the old, cached value (5):

```
// ...
System.out.println(r.lenData()); // => 5

r.setData(new byte[] { 1, 2, 3 });
r.setLenDataRaw(6);
System.out.println(r.lenData()); // => 5 (!)
```

To fix this, you need to call a special method `_invalidate{Inst}` associated with the value instance after changing `len_data_raw`:

```
// ...
System.out.println(r.lenData()); // => 5

r.setData(new byte[] { 1, 2, 3 });
r.setLenDataRaw(6);
r._invalidateLenData();
System.out.println(r.lenData()); // => 3
```

The `_invalidate{Inst}` method invalidates the cached value of the instance so that it is recalculated on the next access.

6.6.4 Lengths and offsets

Current serialization support relies on fixed-length streams, meaning that once a stream is created, it is not possible to resize it later. Therefore, the user will often need to calculate sizes "manually" in their application along with setting the object properties.

6.6.5 Parse instances

Parse (or positional) instances offer a way to parse structures at an arbitrary offset configurable using the `pos` key. They are lazily evaluated, meaning that they are only parsed on the first access. Once they are evaluated, they cache their value and further accesses just return this value.

They have setters and their own `_check{Inst}` method which you should call. Additionally, you can also use a special boolean `set{Inst}_ToWrite` setter, allowing you to disable writing of a specific instance (as `r.set{Inst}_ToWrite(false)` in Java) in a particular KS object. This may be useful for C-style `union` members (several overlapping fields with different types, but only one applies in any object), lookaheads or other positional instances you do not want to write [29].

Parse instances and substreams

By default, the instance is placed in the stream associated with the current user type object that the instance is defined in. The stream can be changed using the `io` key, which can refer to any stream in the object tree.

This is very powerful, but it makes the entire serialization process more complex. Parse instances and its features essentially require the Kaitai serialization to be a 3-pass process; without them, it could have likely been a 1-pass. See Figure 14 for an overview of the 3 passes.

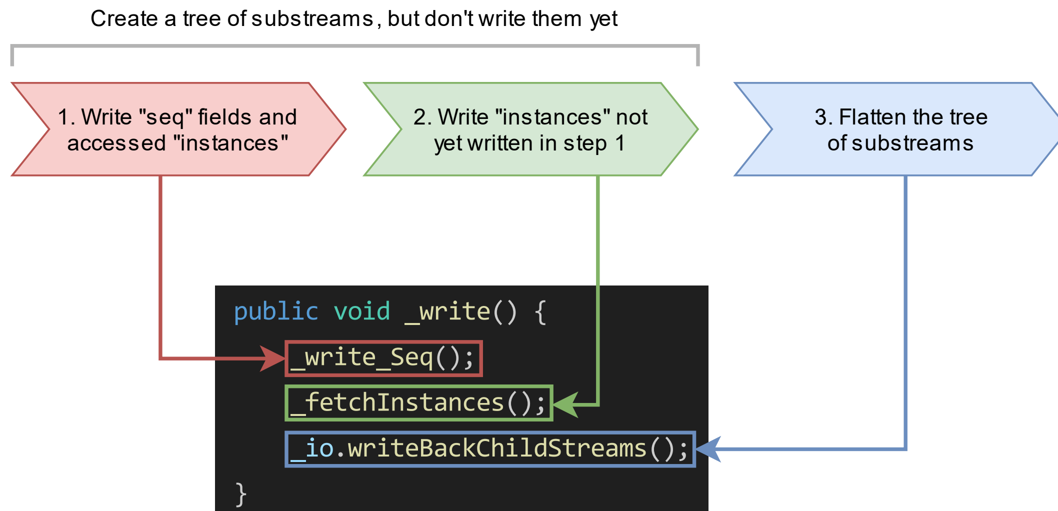


Figure 14: 3 passes of serialization

6.6.6 Parameters

Parameters can be passed to the constructor when instantiating the KS type and you can later change them via setters. Again, KS does not set almost anything automatically, so you are usually in charge of setting all parameters, even though you need to set the parameters to same values that the parent type would pass to them. The `_check` method of the parent type contains checks whether this holds.

Stream parameters

The only parameters you normally do not set are parameters of base type `io` (a KaitaiStream-compatible I/O stream). These are declared as `type: io` or `type: io[]`. They are set automatically by the generated serialization code in child objects (objects with a parent object). However, if your root object has a stream parameter, you have to set it yourself, because Kaitai Struct has no way of knowing what to pass there (the invocation of the root object is obviously not in the .ksy spec) [29].

Streams passed as parameters to the top-level object also require special attention. When you call `r._write()` on the root object `r`, substreams of the `r`'s stream will be collapsed to it. However, this will not happen for the unconnected streams added ex-

ternally via parameters, because they are not in the normal hierarchy of streams under the root stream (and the `_write` method that you call knows directly only about the root stream, so it can only flatten its substreams) [29].

So for every external stream, you have to manually call `extIo.writeBackChildStreams()` after invoking `r._write()` on the root object.

6.6.7 Bit-sized integers

Unlike the existing parser implementation of bit types which relied on explicit `alignToByte()` calls (and this resulted in many problems, because in many cases the compiler failed in where to insert them and where not), all byte-aligned operations in the Java runtime library with serialization support now perform the byte alignment automatically, and the explicit `alignToByte()` calls should not be needed anymore [29]. See section 7.1 for more details.

When you write a structure with X-bit `type: bX` fields, only full bytes are written once they are known. This means that if your format ends at an unaligned bit position, the bits of the final partial byte remain in the internal "bit buffer", but they will not be written to the underlying stream until you do some operation which aligns the position to a byte boundary (e.g. `writeBytes(0)`, `seek(…)`, or explicit `writeAlignToByte()`). However, if you do not have anything else to write and do not need to work with that stream anymore, it is recommended to `close()` the stream, which automatically writes the remaining bits (if any) before closing the stream [29].

This is why you should use the try-with-resources statement to create and manage the stream, as you saw in previous examples:

```
try (KaitaiStream io = new ByteBufferKaitaiStream(output)) {
    hw._write(io);
}
```

It calls `close()` automatically at the end of the try-with-resources block, so you do not have to think about it.

6.6.8 Fields delimited using “size”, “terminator” and “pad-right” keys

Kaitai Struct has several keys to specify how the length of variable-length types should be determined. Variable-length types are byte arrays, text strings and user-defined types in substreams. In contrast, fixed-length types are primitive built-in numeric types (integers in sizes 1, 2, 4 and 8 bytes and floating-point numbers in sizes 4 and 8 bytes). The keys discussed here (like `size` or `terminator`) can be used only for variable-length types, not for fixed-length types.

The `size` key is the simplest. It works in situations where we know the length of a field before we start parsing it. This might be because the size is constant (for example, UUIDs/GUIDs are always 16 bytes long), or the size has already been specified earlier in the format using an integer field (see Figure 15).

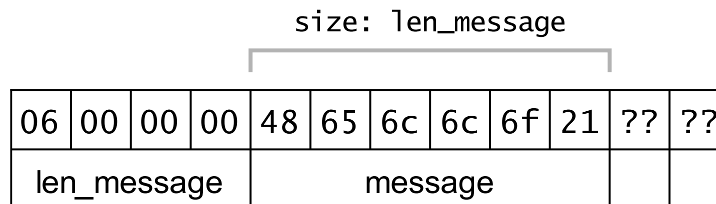


Figure 15: Integer field "len_message" specifying the size of a variable-length field "message"

We use `terminator` for a field whose length we do not know beforehand – we just know where it starts. The length is determined by scanning the bytes until a special byte is reached, which is where the field ends. This is mainly used for null-terminated strings originating from the C language. The byte used as a terminator can be configured with the `terminator` key – see Figure 16.

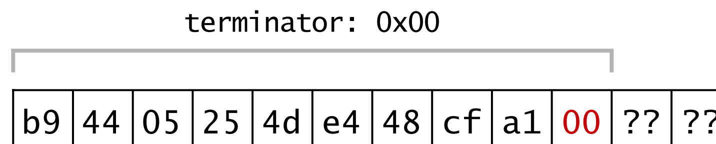


Figure 16: Variable-length field delimited by a terminator byte

The `terminator` key can be used standalone, but also in combination with `size`. In that case, `size` bytes are always reserved for the field contents – the next field in the `seq` structure follows at the offset of this field plus `size`. But the actual contents can be shorter if the `terminator` byte appears within the reserved space. This is most useful in formats with a null-terminated string for which a constant size space is reserved, such as 256 bytes.

The `pad-right` key can be used only for fields with a known reserved size using the `size` key. It specifies that the field byte contents can be shorter than the reserved space; if they are, the rest of the space is padded with a certain byte configured by the value of `pad-right`.

Finally, `pad-right` can be used with `terminator` (resulting in the combination `size` + `terminator` + `pad-right`). `pad-right` usually has no effect on parsing, because the terminator comes before the padding and field contents end once the terminator is encountered (see Figure 17). However, it comes into play if the `terminator` byte is not present in the reserved space. Without `pad-right`, field contents would span the entire reserved space in that case, but with `pad-right`, they may still be shorter if the reserved space contains `pad-right` bytes at the end.

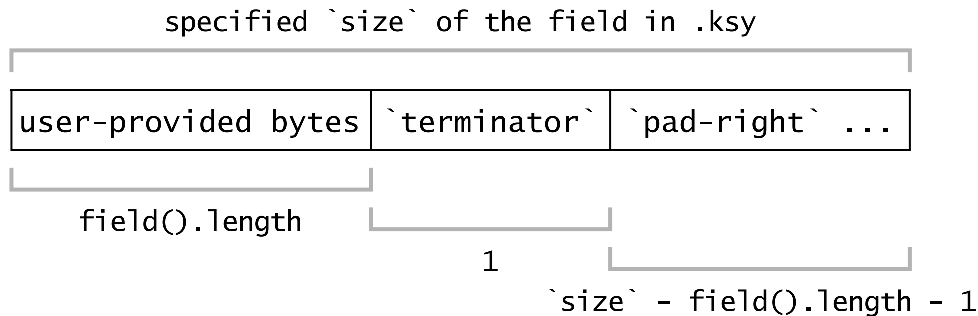


Figure 17: Anatomy of the reserved space of a field using "size", "terminator" and "pad-right" (assuming "include: false", i.e., the default setting)

Consistency checks

The nature of each method of delimiting a field naturally implies properties that values in a successfully parsed tree always satisfy. These properties are exactly the constraints that the data must uphold to be consistent, which is checked by generated consistency checks.

Firstly, if a field only uses `size`, the byte length of its contents must match the `size` exactly, otherwise the data is inconsistent. If a field has `size` with `terminator` and/or `pad-right`, the byte length must be less than or equal to `size`.

Kaitai Struct allows you to specify whether the terminator should be included in the field contents or not using the `include` key. `include: false` is the default value, which omits the terminator from the field contents. `include: true` causes the terminator to be included in the field contents.

The `include` key does not affect parsing (the following fields will be parsed the same regardless of the `include` setting). It only controls what the possible field values are in the object tree, but this is important for consistency checks. `include: false` means that the `terminator` byte must not occur in the byte contents of a field at all. In contrast, `include: true` means that the only occurrence of the `terminator` byte must be at the very end of the byte contents, except for special cases, in which it is allowed to be omitted from the contents. These special cases are such that the field length has reached its upper bound and there is no space left for the terminator. One such case is when the field has `size` specified and the byte length of the field contents is equal to this `size`. Another case is if the field does not use `size`, but its contents extend to the end of stream so that the terminator does not fit. By default, this scenario is treated as an error (normally the terminator is required), but this error can be disabled by specifying `eos-error: false` – then this case is allowed.

Finally, the byte contents of a field using `pad-right` but not using `terminator` must have the last byte (if any) different from `pad-right`. If the field uses `terminator` as well, then it is important to distinguish whether `pad-right` is active or not. `pad-right` is active if it is different from `terminator`. If `pad-right` is equal to `terminator`, then it is inactive, meaning it has effect only on the serialized bytes, not parsing or value consistency – the consistency checks are the same as without `pad-right`. If `pad-right` is

active, the last field byte (if any) must be different from `pad-right` if the `terminator` byte is not present.

6.6.9 Consistency checks that cannot be done in _check()

Sometimes a consistency check cannot be performed in `_check()` because the user expressions from the .ksy specification that the check needs to use or the nature of the check itself do not allow it. A typical example is when the expression makes use of the built-in `_io` variable, which refers to the current I/O stream, for example [29]:

```
seq:
  - id: rest
    size: _io.size - _io.pos
```

Since `rest` is a byte array with the `size` expression denoting its length, it is necessary to check whether the length of this byte array (that might have been changed by the user via a setter) and the value of the `size` expression `_io.size - _io.pos` match. But this expression uses `_io`, so it cannot be performed in `_check()`: `_check()` is meant to check pure data consistency and `_io` may not be available at this point [29]. We can assume the presence of an I/O stream only in `_write()`.

So this consistency check will be moved to `_write()` just before the `rest` field would be written [29]. This ensures that the expression is evaluated in the same context as it would be in `_read()` when parsing, which means we should get the same result as parsing does.

7 Serialization implementation for Java

This chapter discusses the implementation of serialization support in the Java runtime library and the compiler. The implementation follows the design described in chapter 6.

7.1 Runtime library

The most important class of the Kaitai Struct runtime library for Java is `KaitaiStream`. It represents any I/O stream that the generated format modules can operate with. In Java, `KaitaiStream` is an abstract class. There are two implementations: `ByteBufferKaitaiStream` and `RandomAccessFileKaitaiStream`. As their names suggest, they are based on built-in Java classes for working with binary streams, `ByteBuffer` and `RandomAccessFile`.

So far, both `KaitaiStream` implementations have only provided reading functionality, even though both `ByteBuffer` and `RandomAccessFile` support writing as well. So, it was necessary to extend `KaitaiStream` with writing functionality.

This is straightforward for primitive byte-oriented integer and float types (like `u1`, `s2` or `f8`) – for read methods such as `readU1()`, `readS2le()` or `readF8be()`, `write*()` counterparts are added. The counterpart to `readBytes()`, `readBytesFull()` and `readBytesTerm()` is `writeBytes()`. There is also `writeBytesLimit()` used to write bytes of fields with `size` in combination with `terminator` and/or `pad-right`.

For bit-sized integers (`bX` for some bit width `X`), it is a bit more complicated. A long-standing issue also relevant in parsing[4] is the alignment of unaligned bit position to a byte boundary if a byte-oriented operation occurs. Consider the following snippet of a .ksy specification (the stream position at the beginning of `seq` is 0):

```yaml
seq:
  - id: foo
    type: b5
  - id: bar
    type: u1
  - id: baz
    type: b6
```

[4] https://github.com/kaitai-io/kaitai_struct/issues/1070

An alignment operation must be performed between `foo` and `bar`: `foo` leaves 3 bits of byte 0 unparsed, but `bar` must begin at a byte boundary.

The existing approach (still used in Kaitai Struct 0.10, i.e., before serialization) is to insert a call to `alignToByte()` in the generated code, i.e., resolve the alignment at compile time. This has proven to be error-prone and incredibly challenging to get right, because the code responsible for deciding whether an alignment is necessary would have to understand and consider all kinds of control flow statements that the Kaitai Struct language supports, such as `if`, `repeat`, `type/cases`, user-defined types, etc.

The new approach is to resolve the alignment at runtime, i.e., in the runtime library. This is much easier and more reliable because the runtime library knows what exact methods are being called and can react upon that, without having to statically analyze the code that calls them.

If an alignment is missing where it should be during parsing, the internal bit buffer remains in an old state inconsistent with the updated stream position. In the above example, alignment between `foo` and `bar` should discard the 3 unused bits from byte 0 (after `foo` has been parsed) in the bit buffer. If it does not happen, it is not a problem for `bar`, because the type of `bar` is `u1`, which is a byte-oriented type, so it does not use the bit buffer. Therefore, it will be parsed entirely from byte 1 as expected. However, `baz` will be incorrectly parsed – it will use 3 bits left in the buffer and only 3 bits from byte 2, when all 6 bits should come from byte 2. So, missing alignment during parsing only negatively manifests once (if at all) a bit-oriented type comes in the stream afterwards.

The consequence of missing alignment during serialization is more serious. Since the underlying streams used by the `KaitaiStream` class are byte-oriented, there is no way to write individual bits or partial bytes – only whole bytes can be written. This means that writing the 5-bit field `foo` in the above example does not write anything to the underlying byte stream – it only puts these 5 bits into the bit buffer and waits for further operations to determine what the remaining 3 bits of byte 0 should be so that byte 0 can be written. One such operation is bit-byte alignment – if `foo` is followed by the alignment operation as it should be, the 3 bits are determined to be 0 and byte 0 is written. If this alignment is missing, `bar` is written to byte 0 instead – in other words, all byte-oriented fields after `foo` are serialized one byte too early and bit-oriented fields 3 bits too early (at least until bit-byte alignment, if it occurs later in the format), so the serialized output is largely corrupted. Therefore, solving the alignment problem reliably was even more important for serialization than for parsing.

See Figure 18 for an overview of the `KaitaiStream` class method groups after adding writing support.

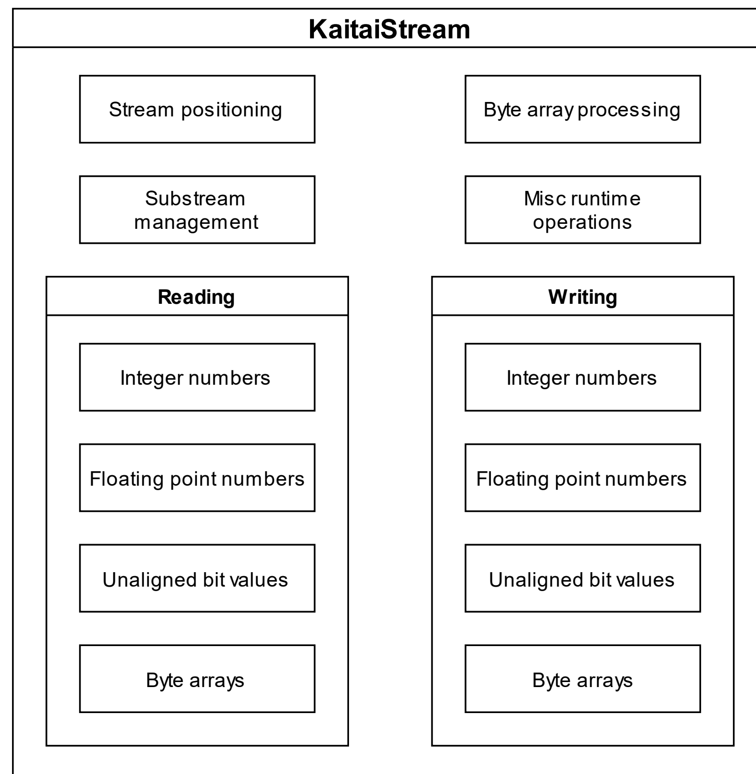


Figure 18: Groups of methods of class KaitaiStream in the Java runtime library

7.2 Compiler

The Kaitai Struct project works on the principle of code generation. Given a KSY specification, Kaitai Struct compiler (KSC) produces source code in the selected language that allows working with binary data in the specified format.

KSC is written in Scala. It can generate parsers in 11 programming languages. Most of its code is independent of the target language. This includes tasks as loading the input KSY specification (technically a YAML file with a certain schema) and pre-compilation (type inference, type checking, resolving names etc.). The final step is the actual compilation, which traverses the KSY object tree and renders source code in the requested target language. This translation is typically handled by a pair of classes, specific for each target language – for example, `JavaCompiler` and `JavaTranslator` in Java. `JavaCompiler` is responsible for the overall structure of the generated code. `JavaTranslator` translates expressions in Kaitai Struct expression language into Java code. Its main method is `translate()`, which receives an AST of the expression and returns a string with the translated expression in Java syntax.

For the generated modules to support serialization in addition to parsing, additional code had to be added to the compiler. This includes new traits: `EveryWriteIsExpression`, `GenericChecks` and `FetchInstances`. These traits are independent of the target language. They are implemented by language-specific compilation classes such as

`JavaCompiler`. Implementors of these traits are required to implement certain methods that add small snippets of code to the generated source code. For example, since `JavaCompiler` wants to implement the `EveryWriteIsExpression` trait, it has to define the `attrPrimitiveWrite()` method, which `EveryWriteIsExpression` calls. Namely `attrPrimitiveWrite()` generates almost `write*()` calls in the generated code. It receives an AST of the expression representing the value to be written, and the data type of the field to write. It inserts a single line of code into the generated output – the statement that calls the appropriate `write*()` method on the `KaitaiStream` object, passing the received value to be written as an argument.

8 Testing

This chapter deals with testing approaches, which were used to verify that the serialization works correctly, i.e., whether it provides the expected results on different inputs. Testing contributes to the reliability and overall quality of the software. It reveals possible regressions in code changes.

Testing was also carried out during development and significantly facilitated development.

8.1 Testing areas

Serialization support in Kaitai Struct consists of two main areas that need testing. One area is to ensure that the writing of consistent data is consistent, i.e., the serialized string will be read back to the same values.

Another area is to ensure that inconsistent data is detected and reported via consistency checks. These checks should be ideally both sound and complete. Sound means consistent data should not be flagged by any check; complete means all possible inconsistencies in data are found.

8.2 Testing write functionality

The basic requirement for a testing method of serialization support is to use as many existing tests as possible that have been developed for testing of parsing. Another goal was to reduce the need to manually write special test code for each test and to modify/adapt these tests.

8.2.1 Method 1: compare serialized output to reference binary files

One method of testing considered was to use assertions in the KST specification to populate the object tree. The serialized output would be compared to the binary file used for parsing. The advantage is that this method does not rely on parsing functionality (so this test works regardless of whether parsing is broken) – it relies on human-supplied reference values. Another advantage is that a direct match of the serialized bytes to the binary is tested. It cannot happen that serialization produces wrong binary output, which the Kaitai Struct parser mistakenly parses to the expected result – incorrect parsing cannot suppress an error in serialization because parsing is not used.

However, in practice, there are two problems with this approach. One problem is that the assertions in KST specifications of existing parsing tests are often incomplete. They cover only a small subset of all values in the object tree. They were not designed to exhaustively describe all values of the object tree so that it can be reconstructed

from the assertions. They were designed only to check that certain few fields under test were parsed correctly. So, they would have to be extended to be complete, which is laborious.

Another problem is the comparison of the serialized output with the reference binary file. In general, the object tree of a format does not necessarily capture full information about every byte of a byte stream. Instead, it only determines a set of byte ranges, eventually bit ranges. This means that if the serialized output is not equal to the full reference binary file, it is not necessarily a serialization failure. It is necessary to know what byte/bit ranges the object tree captures, and compare only those. These ranges would either have to be entered manually in the KST specification, or inferred from parsing (but that introduces a certain dependency on parsing, so the mentioned advantages are reduced to some extent).

Due to these problems, a different approach is used – a write-read roundtrip.

8.2.2 Method 2: write-read roundtrip

The parsing functionality can solve the problem of checking that the serialized output is correct. Instead of directly comparing the serialized output to the expected bitstring, we can parse it to see if we get the object tree that has been serialized. This method is called roundtrip and it is one of the common patterns of property-based testing. It relies on a fundamental requirement for serialization: consistency. If we serialize a consistent object tree, we must be able to fully reconstruct it by deserialization (parsing) based on just the serialized string and the format specification, otherwise serialization is not very useful. So, for an arbitrary consistent object tree, we can test whether serialization can correctly write it by performing a serialization-parsing (or write-read) roundtrip.

As already hinted when discussing method 1, the opposite roundtrip (parsing-serialization) does not work so easily, because parsing can be a lossy process. So, a correctly serialized string generally matches the original binary input only to a certain extent, which makes evaluating the success/failure of serialization more challenging.

We only need a consistent object tree to get the write-read roundtrip started. The easiest way to obtain it is to parse the binary file associated with the existing parsing test, so this is the approach currently used. But it is not the only option – we could also use an object tree that is randomly generated (the problem is that if you generate an object tree completely randomly, most likely it will not be consistent; it takes some effort to develop a generator of consistent object trees).

Note that if the parsing fails (ends prematurely due to an error), roundtrip testing cannot be used, because we do not have a full object tree needed for serialization.

The concrete procedure consists of four phases (see Figure 19). First, the input binary file is parsed according to the format specification – this gives us an object tree with the extracted values. In the second phase, this tree is serialized into a new binary stream. This stream is parsed into a new object tree in the third phase. The fourth phase is the comparison of the data used for serialization with the data after the roundtrip, which is the re-parsed data from the serialized bytes. If serialization works correctly, these two object trees must be the same.

The roundtrip method seems to be a good choice for 99% of cases. For a few specific cases, I used the above-described method 1, i.e., direct comparison of serialized output to expected binary output. In these cases, we want to verify properties of the serialized stream that have no effect on parsing in Kaitai Struct implementation, such as whether a field with `size` and `terminator` is padded correctly (i.e., with the correct byte) after the terminator. Such a field should be padded with zero bytes after the terminator by default unless `pad-right` is specified – then the space after the terminator should be filled with `pad-right` byte.

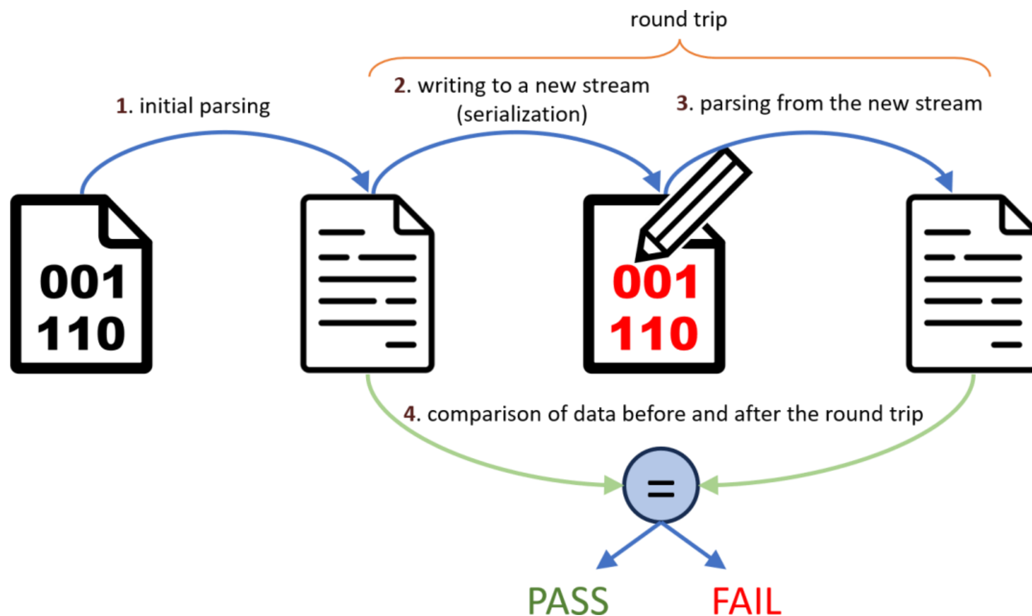


Figure 19: 4 phases of write-read roundtrip

In Java, roundtrip testing is implemented as shown in Figure 20. All test classes for individual test formats extend the `CommonSpec` class. It has two abstract methods that adapt generic roundtrip code to a specific test format: `getStructClass()` and `getSrcFilename()`. The `getStructClass()` method returns the `Class` object of the generated format class, which can be used to create instances of the class and trigger parsing and serialization. The `getSrcFilename()` method returns the name of the binary file associated with the parsing test.

Figure 20: Hierarchy of Java test classes showing the implementation of write-read roundtrip

8.3 Testing consistency checks

In addition to verifying write functionality, it is necessary to verify the functionality of consistency checks. The roundtrip test explained in the previous section only deals with consistent data, it does not attempt to test whether inconsistent data are revealed by consistency checks. The implementation of the roundtrip only looks out for any unsound checks by running all the checks for the object trees it works with, which are expected not to report any errors, so any reported inconsistency is a false positive. But it cannot detect incomplete checks, i.e., when some data inconsistency is not reported. For that, a different set of tests is needed.

Currently, consistency checks are tested on manually specified examples which are expected to pass or fail the check. For example, see the `TestBytesPadTerm` test class:

```
public class TestBytesPadTerm extends CommonSpec {
    // ...

    @Test(expectedExceptions = ConsistencyError.class,
expectedExceptionsMessageRegExp = "Check failed: str_pad,.*")
    public void testCheckLongerStrPad() throws Exception {
        BytesPadTerm r = new BytesPadTerm();
        r.setStrPad("123456789012345678901".getBytes());
        r._check();
    }

    @Test
    public void testCheckGoodLastByteStrPad() throws Exception {
        BytesPadTerm r = new BytesPadTerm();
        r.setStrPad(("@@@@@"+"@@@@@"+"@@@@@"+"@@@@?").getBytes());
        r.setStrTerm("12345678901234567890".getBytes());
        r.setStrTermAndPad("12345678901234567890".getBytes());
        r.setStrTermInclude("12345678901234567890".getBytes());
        r._check();
    }

    @Test(expectedExceptions = ConsistencyError.class,
expectedExceptionsMessageRegExp = "Check failed: str_pad,.*")
    public void testCheckBadLastByteStrPad() throws Exception {
        BytesPadTerm r = new BytesPadTerm();
        r.setStrPad("1234567890123456789@".getBytes());
        r._check();
    }
```

8.4 Evaluation

59 test format specifications were added to the test suite (there were 233, now there are 292 on the "serialization" branch of kaitai_struct_tests), because during the implementation of serialization I came across cases that were not yet covered by the existing tests. Most of the serialization testing is done using a read-write roundtrip. However, it does not test whether inconsistent data that the user might supply is detected, so special test cases for that were added.

I then compiled in read-write mode all the specifications from the format gallery into Java. This also helped me uncover a few generation errors, which I fixed. Additionally, I tested the formats for which I had sample files using the roundtrip mentioned earlier.

9 Conclusion

The addition of serialization to Kaitai Struct was a significant extension to the project that has long been of great interest. Various ideas, analyses, and even implementation efforts have emerged since 2017. The biggest challenge was to create a design that would reflect the large number of possible combinations of KSY language features that appear or could appear in format specifications. This was accomplished. Implemented serialization favors general applicability so that every conceivable format specification is covered. In some cases, this approach requires user interaction with the serialization process in the form of providing additional information or triggering sub-actions. The requirement to be able to use all KSY format specifications written so far without having to change them was fully met. An important part of the project was to design effective test methods early in development, add 59 new tests, and finally thoroughly test the implementation in a variety of ways. Serialization now works for nearly all KS format specifications in the official KS format gallery, of which there are 185 at the time of writing. Links to source code and author's commits can be found in Appendix A.

The serialization code in the KS compiler and runtime library for Java was developed knowing that it will be a template for adding serialization to other programming languages. The result is clean and understandable code. Explanatory comments were added where needed. Also, the documentation on the kaitai.io website is detailed, with installation and usage instructions and explanations of how to handle various cases occurring in the format specifications.

9.1 Effects on Kaitai ecosystem

After the completion of Java serialization, which is the subject of this work, I also added the serialization module for Python.

Furthermore, a working prototype of the C# serialization, modeled after the Java serialization discussed in this thesis, was created by a Canadian developer for the Filestar software by a Swedish company (filestar.com). Filestar is one of the best tools for manipulating data files of diverse formats, it can convert, merge, split, transform, compress them. Serialization in Kaitai Struct for C# has been used there to convert graphic files to the proprietary formats of CorelDRAW (.cdr), Adobe Photoshop (.psd) and other applications.

One of the contributors is now working on serialization in Go, and there is interest in adding serialization to other languages.

Overall, adding serialization has had a great effect on the whole ecosystem. It has greatly expanded the usefulness of Kaitai Struct and has aroused the active interest of many users.

References

[1] Wikipedia contributors. ASCII. In: *Wikipedia, The Free Encyclopedia* [online]. 2024-01-01, 15:46 UTC [cited 2024-01-06]. Available from: https://en.wikipedia.org/w/index.php?title=ASCII&oldid=1193011204

[2] Wikipedia contributors. Unicode. In: *Wikipedia, The Free Encyclopedia* [online]. 2024-01-05, 22:49 UTC [cited 2024-01-06]. Available from: https://en.wikipedia.org/w/index.php?title=Unicode&oldid=1193839471

[3] Wikipedia contributors. UTF-8. In: *Wikipedia, The Free Encyclopedia* [online]. 2024-01-02, 00:24 UTC [cited 2024-01-06]. Available from: https://en.wikipedia.org/w/index.php?title=UTF-8&oldid=1193088259

[4] Wikipedia contributors. Fixed-point arithmetic. In: *Wikipedia, The Free Encyclopedia* [online]. 2023-12-28, 00:04 UTC [cited 2024-01-06]. Available from: https://en.wikipedia.org/w/index.php?title=Fixed-point_arithmetic&oldid=1192165643

[5] Wikipedia contributors. Variable-length quantity. In: *Wikipedia, The Free Encyclopedia* [online]. 2023-08-24, 08:11 UTC [cited 2024-01-06]. Available from: https://en.wikipedia.org/w/index.php?title=Variable-length_quantity&oldid=1171985865

[6] TAURO, Clarence, Nagesswary GANESAN, Saumya MISHRA, and Anupama BHAGWAT. Object Serialization: A Study of Techniques of Implementing Binary Serialization in C++, Java and .NET. In: *International Journal of Computer Applications*, vol. 45 (May 2012), pp. 25-29. ISSN 0975-8887.

[7] MORSCHEL, Lea. *Efficient Message Serialization for Inter-Service Communication in dCache* [online]. Course work [cited 2024-01-06]. Wedel: University of Applied Sciences Wedel, 2019. Available from: https://doi.org/10.3204/PUBDB-2020-01022

[8] VIOTTI, Juan Cruz, and Mital KINDERKHEDIA. *A Survey of JSON-compatible Binary Serialization Specifications* [online]. arXiv preprint, 2022-01-10 [cited 2024-01-06]. Available from: https://doi.org/10.48550/arXiv.2201.02089

[9] *lseek(2)* [online]. Linux man pages. 2023-03-30 [cited 2024-01-06]. Available from: https://man7.org/linux/man-pages/man2/lseek.2.html

[10] MURRELL, Paul. *Introduction to Data Technologies* [online]. 2007-11-20 [cited 2024-01-06]. Available from: https://statmath.wu.ac.at/courses/data-analysis/itdtHTML/node58.html

[11] HUNTER, Geoffrey. *A Comparison Of Serialization Formats* [online]. mbedded.ninja. 2019-01-27 [cited 2024-01-06]. Available from: https://blog.mbedded.ninja/programming/serialization-formats/a-comparison-of-serialization-formats/

[12] ANURADHAC, ARVINDPDMN. *Data Serialization* [online]. Devopedia. 2019-02-27 [cited 2024-01-06]. Available from: https://devopedia.org/data-serialization

[13] *BSON (Binary JSON) Serialization* [online]. 2021-04-29 [cited 2024-01-06]. Available from: https://bsonspec.org/

[14] MessagePack: It's like JSON. but fast and small [online]. 2022-11-29 [cited 2024-01-06]. Available from: https://msgpack.org/

[15] Wikipedia contributors. Data exchange. In: *Wikipedia, The Free Encyclopedia* [online]. 2023-10-30, 16:15 UTC [cited 2024-01-06]. Available from: https://en.wikipedia.org/w/index.php?title=Data_exchange&oldid=1182657896

[16] Wikipedia contributors. Open file format. In: *Wikipedia, The Free Encyclopedia* [online]. 2023-11-23, 22:34 UTC [cited 2024-01-06]. Available from: https://en.wikipedia.org/w/index.php?title=Open_file_format&oldid=1186544091

[17] *FormatFuzzer* [online]. 2022-06-30 [cited 2024-01-06]. Available from: https://uds-se.github.io/FormatFuzzer/

[18] YAKSHIN, Mikhail. *Kaitai Struct: FAQ* [online]. Kaitai Struct: documentation. 2020-09-28 [cited 2024-01-06]. Available from: https://doc.kaitai.io/faq.html

[19] BULSKI, Arkadiusz, Tomer FILIBA, and Corbin SIMPSON. *Introduction* [online]. Construct 2.10 documentation. 2023-12-17 [cited 2024-01-06]. Available from: https://construct.readthedocs.io/en/latest/intro.html

[20] BULSKI, Arkadiusz, Tomer FILIBA, and Corbin SIMPSON. *The Basics* [online]. Construct 2.10 documentation. 2023-12-14 [cited 2024-01-06]. Available from: https://construct.readthedocs.io/en/latest/basics.html

[21] MENDEL, Dion. *dmendel/bindata Wiki* [online]. GitHub. 2023-06-25 [cited 2024-01-06]. Available from: https://github.com/dmendel/bindata/wiki/Home/b44eb5108109fa0b32cece7f056baf07774fc75b

[22] COSTELLO, Roger L. *Describing Data Formats using DFDL, Part I* [online]. xFront. 2019-12-23 [cited 2024-01-06]. Available from: https://www.xfront.com/DFDL/DFDL-part1.pptx

[23] Wikipedia contributors. Shapefile. In: *Wikipedia, The Free Encyclopedia* [online]. 2023-08-27, 21:30 UTC [cited 2024-01-06]. Available from: https://en.wikipedia.org/w/index.php?title=Shapefile&oldid=1172560051

[24] LAWRENCE, Steve. *shape.dfdl.xsd in DFDLSchemas/shapeFile* [online]. GitHub. 2019-06-26 [cited 2024-01-06]. Available from: https://github.com/DFDLSchemas/shapeFile/blob/25b7e7db28d6398a1e7045144c4ecb856b21a4ab/src/main/resources/org/mitre/shp/xsd/shape.dfdl.xsd

[25] Wikipedia contributors. Data Format Description Language. In: *Wikipedia, The Free Encyclopedia* [online]. 2023-06-30, 14:02 UTC [cited 2024-01-06]. Available from: https://en.wikipedia.org/w/index.php?title=Data_Format_Description_Language&oldid=1162672081

[26] VITUCCI, Nicola. *Apache Daffodil* [online]. APOTHEM | Apache Project(s) of the month. 2019-04-07 [cited 2024-01-06]. Available from: https://apothem.blog/apache-daffodil.html

[27] LAWRENCE, Steve. *Unparser Overview* [online]. Apache Daffodil. 2018-02-07 [cited 2024-01-06]. Available from:

https://cwiki.apache.org/confluence/display/DAFFODIL/Unparser+Overview

[28] YAKSHIN, Mikhail. ftl_dat.ksy in kaitai-io/kaitai_struct_formats [online]. GitHub. 2019-01-02 [cited 2024-01-06]. Available from: https://github.com/kaitai-io/kaitai_struct_formats/blob/83886d266995b31a233127f2b049bca311a6348b/game/ftl_dat.ksy

[29] PUCIL, Petr. *Serialization Guide* [online]. Kaitai Struct: documentation. 2023-10-23 [cited 2024-02-15]. Available from: https://doc.kaitai.io/serialization.html

[30] PUCIL, Petr. *Kaitai Struct — serialization* [online]. GitHub Gist. 2023-12-29 [cited 2024-02-15]. Available from: https://gist.github.com/generalmimon/fc22e97faf1fe4b4edc8279b0caa152d/a93bec57b1554651b7fcc7cd1eec002eef3fa76a

Appendix A: Source code

All code is hosted in public repositories on GitHub in the kaitai-io organization.

- Kaitai Struct compiler with serialization support: https://github.com/kaitai-io/kaitai_struct_compiler/tree/4066d1e2d71238b2f37bd6a75e20b1271b7ae14d
- Kaitai Struct runtime library for Java with serialization support: https://github.com/kaitai-io/kaitai_struct_java_runtime/tree/ebace6b4adbb38d3990089dfd8ee680ed86490de
- Kaitai Struct tests: https://github.com/kaitai-io/kaitai_struct_tests/tree/629484b021cf5835e9bfee40bc621f0108120b7c
- Serialization Guide in Kaitai Struct documentation: https://github.com/kaitai-io/kaitai_struct_doc/blob/3ff727ff924bd661dcf1514b6650338852d92993/serialization.adoc

Author's commits (GitHub profile: https://github.com/generalmimon) at revisions with serialization support:

- Kaitai Struct compiler: https://github.com/kaitai-io/kaitai_struct_compiler/commits/4066d1e2d71238b2f37bd6a75e20b1271b7ae14d/?author=generalmimon
- Kaitai Struct runtime library for Java: https://github.com/kaitai-io/kaitai_struct_java_runtime/commits/ebace6b4adbb38d3990089dfd8ee680ed86490de/?author=generalmimon
- Kaitai Struct tests: https://github.com/kaitai-io/kaitai_struct_tests/commits/629484b021cf5835e9bfee40bc621f0108120b7c/?author=generalmimon
- Serialization Guide in Kaitai Struct documentation: https://github.com/kaitai-io/kaitai_struct_doc/commits/3ff727ff924bd661dcf1514b6650338852d92993/serialization.adoc?author=generalmimon