USING BITSTREAM SEGMENT GRAPHS FOR COMPLETE DESCRIPTION OF DATA FORMAT INSTANCES

Michael Hartle, Friedrich-Daniel Möller, Slaven Travar, Benno Kröger and Max Mühlhäuser
Telecooperation, Technische Universität Darmstadt, Hochschulstr. 10, D-64289 Darmstadt, Germany

Keywords: Software engineering, data format, bitstream, complete description.

Abstract: Manual development of format-compliant software components is complex, time-consuming and thus error-prone and expensive, as data formats are defined in semi-formal, textual specifications for human engineers. Existing approaches on a formal description of data formats remain at high-level descriptions and fail to describe phenomena such as compression or fragmentation that are especially common in Multimedia file formats. As a step-stone towards the description of data formats as a whole, this paper presents *Bitstream Segment Graphs* as a complete model on data format *instances* and presents an example PNG where a complete model on data format instances is required.

1 INTRODUCTION

A data format is an abstract concept for expressing how some information is laid out in terms of bits and bytes. Let us assume we want to know the width of a Portable Network Graphics (PNG) image file. The PNG file format (Duce, 2003) defines the overall structure, assigns the meaning of image width to some bit range and maps these specific bits to a typed literal. Using this knowledge, we know how to find out the width of an image stored in a PNG file and can write some code similar to Figure 2.

Now, when we implement an algorithm that works on data following a data format, we translate data format knowledge into source code. The resulting source code for format-compliant processing strongly depends on initial factors such as its purpose and aspects of the target environment. Such aspects involve the underlying hardware, the operating system, the programming language, the style of programming, the APIs we intend to use, how we want to keep the information in memory, whether or not the data format is octet-aligned and so on. If we change one initial factor, eg. switch from Java to C++, PHP or VHDL, the result differs substantially in terms of software design and implementation. Once designed and implemented, adapting source code to change factors is a labourous manual task that is often undesirable.

1.1 Problem

In practice, current data formats in domains like Multimedia have reached a tremendous complexity, which can be demonstrated with an example. For writing software that is natively capable of setting up a H.323 video conference call to a Microsoft NetMeeting software client or a Sony hardware client, we needed to implement the H.225 and H.245 protocols besides others. The data format of their protocol messages is defined using the Abstract Syntax Notation One (ASN.1) (ITU-T, 1997) and the ASN.1 PER encoding (ITU-T, 2002) in their respective specifications, all in all involving several hundreds of pages basically intended for human engineers.

Fortunately, formal ASN.1 specifications of H.225 and H.245 exist that can be translated into source code using appropriate ASN.1 compilers. For an object-oriented Java implementation capable of sending and receiving the respective protocol messages, we did so using openASN1 (Hoss and Weyland, 2007). The translation resulted in 321 Java classes with 865kb of source code for H.225 and in 1.005 Java classes with 3.87mb of source code for H.245. So without focusing too much on the numbers, this example makes obvious that manually translating data format knowledge into source code is all but a trivial task for human engineers that make mistakes. For data formats in general, this manual translation process is complex, time-consuming and thus error-prone and expensive. Unfortunately, ASN.1 is not generally applicable and

Hartle M., Möller F., Travar S., Kröger B. and Mühlhäuser M. (2008).
USING BITSTREAM SEGMENT GRAPHS FOR COMPLETE DESCRIPTION OF DATA FORMAT INSTANCES.
In *Proceedings of the Third International Conference on Software and Data Technologies - ISDM/ABF*, pages 198-205
DOI: 10.5220/0001891101980205
Copyright © SciTePress

up till now, related work in literature does not deliver something comparable for data formats in general, as we will see later in section 2.

Moreover, the initial factors of a translation often do not remain constant. Data formats are revised, get extended or have to be adapted to be interoperable with deviating implementations. Instead of just reading data, we may need to write it as well. We may have to port parts of it to a J2ME mobile phone, or an embedded system. We may even need to change the programming language for better integration with other components. With every change, we again have to heavily adapt previous source or even retranslate, using manual labour.

Automating the translation process of data format knowledge to source code thus becomes attractive. For that purpose, we need explicit data format knowledge to be present in a machine-processable sense. Yet, no suited model on data formats exists in literature, so for data format knowledge, we are currently limited human processing of semi-formal, textual specifications or source code of other implementations.

1.2 Restating the Problem

The intention of a data format is to provide a lossless transport representation of information for the purpose of storage and transmission for each of its data format instances. A *data format instance* is a bijective mapping between structured, semantic literals as information and a finite sequence of bits, or *bitstream*, as transport representation, with an example shown in Figure 1. A *data format* is a finite definition of a possibly infinite set of data format instances, which is analogous in definition to that of a formal language (Mateescu and Salomaa, 1997).

Existing research in literature does not provide a complete, generally applicable and machine-processable model for describing arbitrary data formats such as those in practice. Just like a data format is composed from its instances, we need to be able to describe these instances before we can describe a data format as a whole. As such a model on data format instances is a necessary prerequisite which does not exist as well, it is the main subject of this paper.

1.3 Implications

Due to the lack of suited models, common engineering tasks with respect to data formats basically remain manual processes for human engineers that make mistakes, which introduces follow-up problems:

- Defining or reverse-engineering of a data format has no standard representation suited for automated documentation and exchange. Ensuring correctness, completeness, consistency and unambiguousness of semi-formal, textual specifications is entirely in the hand of human judgement and therefore hard to guarantee.
- Designing and implementing format-compliant components for typical purposes such as parsing, in-memory representation and serialization for a specific environment is a complex task in itself. Since the complexity of a data format is usually present in its design and implementation as well, this manual task becomes even more complex and therefore error-prone. Until discovered and patched, errors in the implementation can lead to security issues such as buffer overflows or break interoperability with other implementations which is often hard to attribute to. Moreover, the resulting implementation is bound to its intended data format, environment and purpose, where any non-trivial change to any of these requires substantial adaptation or even redevelopment. It can be assumed that the arising development cost limits the diversity of existing, reusable implementations.
- As long as access to and navigation of data in a specific data format directly depends upon the existance of suited format-compliant implementations that have to be developed manually, data remains tightly coupled with these implementations. This is a hard problem for *Digital Preservation* efforts of libraries, as the obsolescence of applications over time threatens a large body of digitally born data (Ross and Hedstrom, 2005) on an individual, corporate and national scale (Wettengel, 1998), much of which comprises our digital cultural heritage.

1.4 Contribution

In this paper, we present *Bitstream Segment Graphs* (BSG) as a complete, generally applicable and machine-processable model on *data format instances*, serving a step-stone towards a later corresponding model on data formats as a whole.

We start by taking a look on related work in literature (Section 2) and develop a more distinct notion of completeness (Section 3). Based on that notion, the formalism of Bitstream Segment Graphs is defined (Section 4.1) and an algorithm for its composition is given, together with a visual representation is given (Sections 4.2 and 4.3). Using our model, we finally present a practical example (Section 5) that ex-

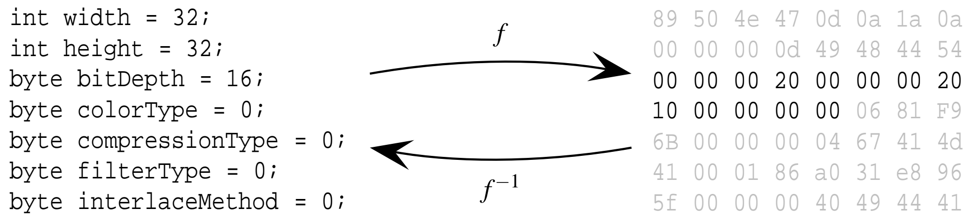

Figure 1: A *data format instance* is a bijective mapping between a set of semanic literals (left) and a finite sequence of bits, depicted as f, where the bit sequence is shown as hexdump (right) with the relevant segment colored in black. The presented excerpt is the IHDR structure of the PNG image "oi2n0g16.png" (van Schaik, 1998).

```
// Write image header data
bufferOut.writeLong(width);
bufferOut.writeLong(height);
bufferOut.write(bitDepth);
bufferOut.write(colorType);
bufferOut.write(compressionMethod);
bufferOut.write(filterMethod);
bufferOut.write(interlaceMethod);
buffer = bufferOut.toByteArray();

// Write IHDR chunk
out.writeLong(buffer.length);
out.writeLong(0x49484454);
out.write(buffer);
out.writeLong(calcCRC(buffer));
```

```
// Read chunk header
length = in.readLong();
type = in.readLong();
if (type == 0x49484454) {
 in.readBuffer(buffer, 0, length);
 crc = in.readLong();
 if ((length == 13) && (crc ==
calcCRC(buffer)) {
   tempIn = new ByteArrayInputStream(buffer);
   bufferIn = new DataInputStream(tempIn);

   // Read image header data
   width = bufferIn.readLong();
   height = bufferIn.readLong();
   bitDepth = bufferIn.read();
   colorType = bufferIn.read();
   compressionType = bufferIn.read();
   filterType = bufferIn.read();
   interlaceMethod = bufferIn.read();
 }
}
```

Figure 2: Example Java source code extracts for serializing (left) and parsing (right) a PNG IHDR structure which implements data format knowledge.

isting approaches cannot describe completely and finally end the paper with conclusions (Sections 6) and acknowledgements.

2 RELATED WORK

The primary source of research on data format description originates from the field of Multimedia, where format-unaware, "dumb" delivery of multimedia data becomes a problem in the face of widely heterogenous environments. Network bandwidth and/or latency may not allow timely delivery of multimedia data to the end-user. Devices may lack computational power for timely decoding or possess only a display with limited screen resolution, which cannot make full use of the encoded video. Research on *Universal Media Access (UMA)* addresses this problem using format-aware on-the-fly content adaptation and filtering.

UMA thus lead to research on data formats and their specification, which resulted in *MSDL-S* (Eleftheriadis, 1996) for the documentation of data structures, its successors *Flavor* and *XFlavor* (Eleftheriadis and Hong, 2004) for the automated generation of format-compliant software components or the *Bitstream Syntax Description Language* (BSDL) (Amielh and Devillers, 2001) recombinations and extensions like *gBSDL* (Vetro et al., 2006), *BFlavor* (De Neve et al., 2006) and *gBFlavor* (Deursen et al., 2007) for high-level multimedia content adaptation and filtering.

Upon closer examination, these approaches focus on a limited, typically high-level description of data and do not describe transformations that may be present in data format instances such as compression

or fragmentation. These approaches therefore do not provide a model which is sufficient to describe arbitrary data formats and their instances to completion.

3 ANALYSIS

For an approach on data format description to be generally applicable in a real-world scenario, requirements need to be fulfilled. First of all, we need to describe bitstreams of arbitrary partitioning, alignment and length in order to cover everything. Furthermore, to describe the bijective mapping between the bitstream and its contained information, we need to describe its composition. These requirements can be understood as varying degrees of completeness regarding the description of a data format instance and can be restated as follows:

- *Width-completeness* is given if a data description can cover arbitrary bitstreams. For general applicability, it mandates bit granularity and support for both arbitrary alignment and length for its descriptive means.
- *Depth-completeness* is given if a data description is width-complete and can provide for a bijective mapping between the bitstream and its contained information in the form of structured, semantic, independent literals. It mandates the existence of suited descriptive means for arbitrary transformations on bitstreams and the encoding of literals.

The previously identified problem of existing approaches on the description of data format instances can now be restated as a lack of depth-completeness, especially regarding block and concatenating transformations. These are required eg. for handling zlib-compressed bitstream segments in PNGs or interleaved audio/video bitstream fragments in many multimedia containers such as MPEG-4 or transport streams such as MPEG-2 TS.

4 MODEL

We now introduce Bitstream Segment Graphs as a generally applicable model on data format instances which is width- and depth-complete.

4.1 Definition

The following definitions include the term *bitstream* to make their scope explicit. Whenever no ambiguity is introduced, it may be omitted otherwise.

Definition 4.1 (Bitstream Segment). *Given a bitstream segment $v \in V$, the set of bitstream segments V, the set of finite consecutive bit sequences $B = \{0,1\}^n, n \in \mathbb{N} \setminus \{0\}$ and $\varphi : V \mapsto B$, then the bitstream segment v represents a finite consecutive bit sequence $\varphi(v) \in B$.*

Definition 4.2 (Bitstream Source). *A bitstream source is a root bitstream segment $v_{Root} \in V$ with a defined $\varphi(v_{Root})$.*

A bitstream source represents a digital item which is composed according to a data format. Files, network packets or file systems on some storage medium are examples for octet-aligned bitstream sources.

Definition 4.3 (Bitstream Encoding). *Given a bitstream encoding $e = (rel, v, l) \in R_E, v \in V, l \in L$, where R_E is the set of bitstream encodings and L is the set of literals, then for a given v, e specifies a mapping relation $rel(\varphi(v), l)$, required to be bijective. It is abbreviated with $\phi(v) = l$, where $\phi : V \mapsto L$.*

A bitstream segment can represent an encoded literal that is part of the data contained in a bitstream source. For example, there are two bitstream segments within a PNG file which contain encoded integers that represent the width and height of the image.

Definition 4.4 (Bitstream Transformation). *Given a bitstream transformation $t = (rel, V_{in}, V_{out}, P) \in R_T$, where V_{in}, V_{out} are totally ordered sets with $V_{in} \subset V, V_{out} \subset V, V_{in} \neq \emptyset, V_{out} \neq \emptyset, V_{in} \cap V_{out} = \emptyset$, R_T is the set of bitstream transformations and P is the set of parameters, then t specifies a mapping relation $rel(V_{in}, V_{out}, P)$, required to be bijective, between V_{in} and V_{out} under application of P.*

In general, a bitstream transformation t bijectively maps a set of input bitstream segments V_{in} to a a set of new bitstream segments V_{out} as result of the transformation. *Normalized bitstream transformations* categorized by $|V_{in}| : |V_{out}|$ cardinality are

- the concatenating transformation of multiple fragment segments into one composite segment ($m:1$),
- a class of block transformations such as decompression or decryption ($1:1$) and
- segmenting transformation of a structured segment into multiple separate bitstream segments ($1:n$).

Arbitrary transformations of $m:n$ cardinality can be composed by concatenating two or more normalized transformations.

Definition 4.5 (Bitstream Segment Graph). *Given a set of bitstream transformations R_T and a set of*

bitstream encodings R_E, then R_T and R_E induce a bitstream segment graph (BSG). It is a weakly connected, directed acyclic rooted graph $G = (V,E)$ with a set of bitstream segments V as vertices and a set of directed edges $E \subset V \times V$, connecting transformation input/output pairs of bitstream segments. A BSG describes the composition of a bitstream source and is complete iff

$$\forall v \in V\colon (\exists! t = (rel_t, V_{in}, V_{out}, P) \in R_T, v \in V_{in}) \oplus (\exists! e = (rel_e, v_e, l) \in R_E, v = v_e)$$

A BSG is composed from bitstream transformations and encodings, which are required to have bijective mapping relations. It therefore provides a bijective mapping between its bitstream source and its contained literals and thus satisfies the depth-completeness requirement.

Definition 4.6 (Transformation Dependency). *A transformation dependency exists if for a bitstream segment v there exists a nonempty set of bitstream segments $\varpi(v) \in 2^V$ with $t = (rel, V_{in}, V_{out}, P) \in R_T, v \in V_{out}$ that v depends on, where $\varpi : V \mapsto V^n, n \in \mathbb{N}$.*

Definition 4.7 (Functional Dependency). *A functional dependency of a bitstream segment $v \in V$ on a nonempty set of bitstream segments $V_{dep} \subset V$ with $v \notin V_{dep}$ exists if the data format defines a function f and mandates that $\phi(v) = f(V_{dep})$.*

An example of both a transformation dependency and a functional dependency is a bitstream segment which encodes the variable length of another bitstream segment. For extracting the latter from a segmentation, the value of the former is required as a parameter to the transformation. Another example of a functional dependency is a Cyclic Redundancy Code (CRC) on a set of bitstream segments, stored in another bitstream segment.

Transformation and functional dependencies put constraints on possible orders of processing for bitstream segments and validity that format-compliant software implementations of parsers, object / streaming models and serializers need to handle in their operation.

4.2 Composition Algorithm

Using definitions 4.1 to 4.5, we are able to describe the bijective mapping between a bitstream source and its set of contained literals. The following simple algorithm constructs a BSG step-by-step. For a construction at step x, the tuple

$$(v_{Root}, V_x, V_{leaf_x}, V_{literal_x}, R_{T_x}, R_{E_x})$$

describes a designated root bitstream segment v_{Root}, a set of bitstream segments V_x, a set of leaf bitstream segments V_{leaf_x}, a set of literal bitstream segments $V_{literal_x}$, a set of bitstream transformations R_{T_x} and a set of bitstream encodings R_{E_x}, whereas initial values are

$$\begin{aligned} V_0 &= \{v_{Root}\} \\ V_{leaf_0} &= \{v_{Root}\} \\ V_{literal_0} &= \emptyset \\ R_{T_0} &= \emptyset \\ R_{E_0} &= \emptyset \end{aligned}$$

Starting at step $x = 1$, each step either adds a transformation or an encoding. For a transformation, the addition of $t = (rel, V_{in}, V_{out}, P) \notin R_{T_{x-1}}$, $V_{in} \subseteq V_{leaf_{x-1}}$ results in

$$\begin{aligned} V_x &= V_{x-1} \cup V_{out} \\ V_{leaf_x} &= V_{leaf_{x-1}} \cup V_{out} \setminus V_{in} \\ V_{literal_x} &= V_{literal_{x-1}} \\ R_{T_x} &= R_{T_{x-1}} \cup \{t\} \\ R_{E_x} &= R_{E_{x-1}} \end{aligned}$$

whereas the addition of an encoding $e = (rel, v, l) \notin R_{E_{x-1}}, v \in V_{leaf_{x-1}}$ results in

$$\begin{aligned} V_x &= V_{x-1} \\ V_{leaf_x} &= V_{leaf_{x-1}} \setminus v \\ V_{literal_x} &= V_{literal_{x-1}} \cup \{l\} \\ R_{T_x} &= R_{T_{x-1}} \\ R_{E_x} &= R_{E_{x-1}} \cup \{e\} \end{aligned}$$

For step y, the tuple induces a BSG $G_y = (V_y, E_y)$ where E_y is defined as follows:

$$\forall t = (rel, V_{in}, V_{out}, P) \in R_{T_y},$$
$$\forall v_s \in V_{in}, \forall v_t \in V_{out} : e = (v_s, v_t) \in E_y$$

These steps are repeated until $V_{leaf_z} = \emptyset$, where the algorithm terminates as no further addition of either transformation or encoding to leaf bitstream segments is possible.

4.3 Visual Representation

For the representation of a BSG, bitstream segments are categorized into *types*, based on normalized transformations and encodings as shown in Table 1. To prevent a conflicting type assignment for bitstream segments that have both the “upward” composite type and another “downward” type, an identity transformation is inserted after the composite and the “downward” type is assigned to the newly inserted bitstream segment.

Table 1: Types of bitstream segments.

Leaf	Bitstream segment participates in Encoding	Transformation	Type
yes	no	no	Generic
yes	any	no	Primitive
no	no	segmentation input	Structure
no	no	transformation input	Transcode
no	no	concatenation input	Fragment
no	no	concatenation output	Composite

Depending on their type, segments are depicted as shown in Figure 3, where *start* and *end* denote inclusive start and exclusive end bit positions relative to the parent bitstream segment(s), *type* denotes the bitstream segment type, *parameter* denotes a parameter for some types and *id* denotes some plaintext identification.

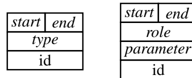

Figure 3: Visual representations; generic, structure and composite bitstream segments (left); fragment, primitive and transcode bitstream segments (right).

Besides the visual representation, we have defined an RDF ontology based on bitstream segment types for storage, processing and interchange of bitstream segment graphs for arbitrary data, and implemented a Java-based annotation tool for their construction. Both the RDF ontology and the annotation tool are subject of another publication.

5 EXAMPLE

The PNG Test Suite (van Schaik, 1998) includes the file *oi2n0g16.png* which contains zlib-compressed grayscale image data in the form of filtered scanlines, fragmented in two separate chunks. The file was selected as its composition contains both a block transformation and a concatenation transformation, which other approaches cannot describe completely due to a lack of depth-completeness. Figure 4 shows a partial bitstream segment graph with a depth-complete description of the contained pixel data including the required two segmentations, concatenation, decompression and PNG-specific filtering block transformation. For the sake of readability, the remaining more simple structures *Signature*, *IHDR*, *gAMA* and *IEND* are shown in a collapsed state only as they just contain several primitive segments.

$$\begin{aligned}
\text{File} &\rightarrow \text{IDAT \#1}, \text{IDAT \#2} \\
\text{IDAT \#n} &\rightarrow \text{Len \#n}, \text{Type \#n}, \text{Data \#n}, \text{CRC \#n} \\
\text{Data \#1}, \text{Data \#2} &\rightarrow \text{Composite} \\
\text{Compressed} &\rightarrow \text{Scanlines} \\
\text{Scanlines} &\rightarrow \text{Pixels}
\end{aligned}$$

An example for both a transformational dependency and a functional dependency is the segmentation of IDAT #1 and IDAT #2 as the length of Data #1 and Data #2 depends on $\phi(\text{Len \#1})$ and $\phi(\text{Len \#2})$ respectively. Examples of functional dependencies are CRC #1 and CRC #2, as $\phi(\text{CRC \#n})$ depends on $\phi(\text{Type \#n})$ and $\phi(\text{Data \#n})$.

6 CONCLUSIONS

6.1 Summary

This paper has given an in-depth introduction on needs of research on data format description. It linked the description of data formats and data format instances and introduced the terminology of depth-complete and dependency-complete descriptions. Based on this terminology, the paper contributed the Bitstream Segment Graph as a formalism for depth-complete data format instance descriptions and defined a corresponding visual representation. Using a real-world PNG from a test suite, we have shown an example that requires depth-completeness and presented its BSG representation.

Although Bitstream Segment Graphs were developed for the description of data formats as a whole, a model on data format instances has merits on its own. It allows to document the composition of data and thus serves well during reverse-engineering of instances of unpublished data formats used in protocols and file formats. A practical example is the reverse-engineering of exploits in IT Security using BSGs (Hartle et al., 2008) in order to understand its mechanisms and patch vulnerable implementations.

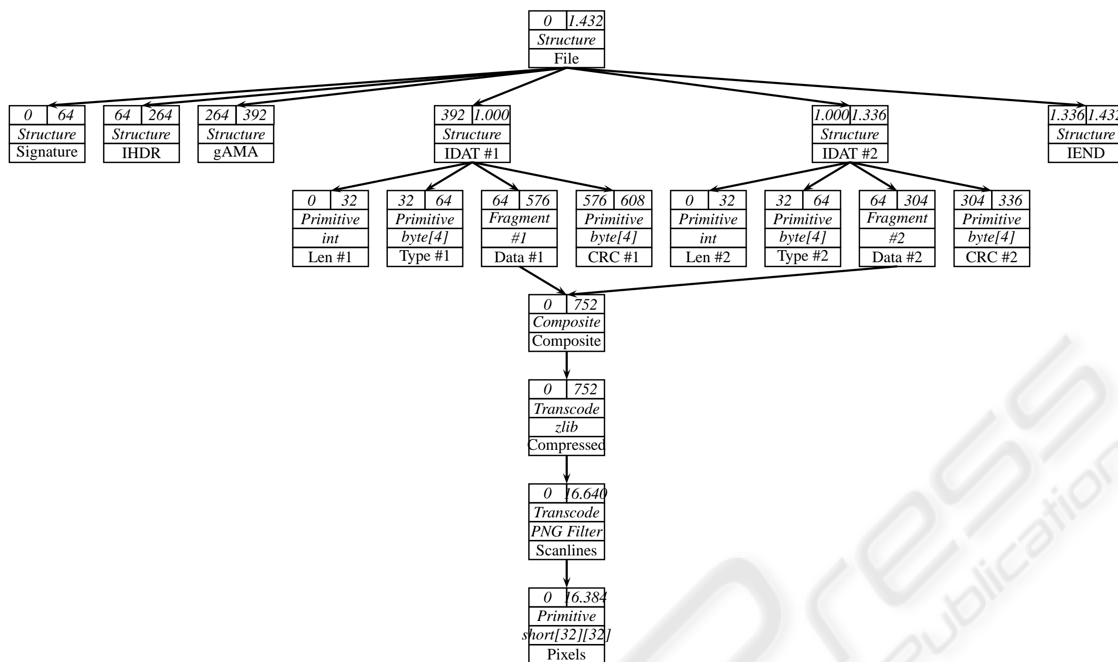

Figure 4: Partial bitstream segment graph for file “oi2n0g16.png”, showing the bijective mapping of two PNG IDAT chunks to a 16 bit grayscale image with a resolution of 32×32 pixel.

6.2 Outlook

Further research on data format instance description such as the application of machine learning are currently underway and will hopefully result in a model for describing arbitrary data formats. Once data formats can be described completely in a formal way, processing and applying data format knowledge in an automated manner should help simplify the development of format-compliant software components and remove the tight coupling of data and applications.

ACKNOWLEDGEMENTS

The authors would like to thank Tobias Klug, Guido Rling and Gina Hue for providing feedback on drafts as well as Clayton Hoss and Marc Weyland for developing openASN1 as their diploma thesis.

REFERENCES

Amielh, M. and Devillers, S. (2001). Multimedia Content Adaption with XML. In *8th International Conference on Multimedia Modelling*, pages 127–145.

De Neve, W., Van Deursen, D., De Schrijver, D., Lerouge, S., De Wolf, K., and Van de Walle, R. (2006). BFlavor: A harmonized approach to media resource adaptation inspired by MPEG-21 BSDL and XFlavor. *EURASIP Signal Processing: Image Communication*, 21(10):862 –889.

Deursen, D. V., Neve, W. D., Schrijver, D. D., and de Walle, R. V. (2007). Automatic generation of generic Bitstream Syntax Descriptions applied to H.264/AVC SVC encoded video streams. *iciap*, 0:382–387.

Duce, D. (2003). Portable Network Graphics (PNG) Specification (Second Edition): Information technology – Computer graphics and image processing – Portable Network Graphics (PNG): Functional specification. ISO/IEC 15948:2003 (E).

Eleftheriadis, A. (1996). The Benefits of Using MSDL-S for Syntax Description. Contribution ISO/IEC JTC1/SC29/WG11 MPEG96/M1555.

Eleftheriadis, A. and Hong, D. (2004). Flavor: a formal language for audio-visual object representation. In *MULTIMEDIA ’04: Proceedings of the 12th annual ACM international conference on Multimedia*, pages 816–819, New York, NY, USA. ACM Press.

Hartle, M., Schumann, D., Botchak, A., Tews, E., and Mhlhuser, M. (2008). Describing Data Format Exploits using Bitstream Segment Graphs. Proceedings of The Third International Multi-Conference on Computing in the Global Information Technology ICCGI 2008.

Hoss, C. and Weyland, M. (2007). openASN.1: Entwicklung und Evaluation eines ASN.1-Compilers und PER-Codecs unter Java. Diploma thesis, Reliable Basic Support group, Depart-

ment of Computer Science, TU Darmstadt. http://www.openasn1.de/media/documents/Diplomarbeit-openASN.1-0.5.0b.pdf, last accessed 2008-04-02.

ITU-T (1997). Recommendation X.680 (12/97) — Abstract Syntax Notation One (ASN.1): Specification of Basic Notation. ITU-T, Geneva.

ITU-T (2002). Recommendation X.691 (07/02) — ASN.1 Encoding Rules: Specification of Packed Encoding Rules (PER). ITU-T, Geneva.

Mateescu, A. and Salomaa, A. (1997). *Formal Languages: an Introduction and a Synopsis*, chapter 1, pages 1–40. Springer Verlag.

Ross, S. and Hedstrom, M. (2005). Preservation research and sustainable digital libraries. *Int. J. on Digital Libraries*, 5(4):317–324.

van Schaik, W. (1998). PngSuite - the official set of PNG test images. http://www.schaik.com/pngsuite/pngsuite.html, last accessed 2008-01-02.

Vetro, A., Timmerer, C., and Devillers, S. (2006). *The MPEG-21 Book*, chapter Digital Item Adaptation - Tools for Universal Multimedia Access, pages 243–281. John Wiley and Sons Ltd.

Wettengel, M. (1998). *German Unification and Electronic Records*, chapter 18, pages 265–276. Oxford University Press. ISBN 0198236336.

SciTePress
Science and Technology Publications