A Logic-Based Approach to the Formal Specification of Data Formats

Michael Hartle, Arsene Botchak, Daniel Schumann, Max Mühlhäuser

Technische Universität Darmstadt
Telecooperation Group
Hochschulstr. 10
D-64289 Darmstadt, Germany
{mhartle,max}@tk.informatik.tu-darmstadt.de

Abstract

Processing information stored as data in a specific data format is tightly coupled with software implementations that handle necessary elementary processes such as reading and writing. These implementations depend on specific technological environments and thus age due to rapid technological change. The resulting effective loss of information is a major problem for Digital Preservation.

In order to provide for persistent, authentic access to stored information, this paper presents a logic-based approach for the formal specification of data formats.

Introduction

What turns data into information is the knowledge on its semantics, its intended meaning. If this knowledge is lost, so is our access to information that is contained in data. A good example from history was the inability to read ancient hieroglyphic Egyptian script for more than a millennium, fortunately solved by the happenstance of the Rosetta Stone. Only by the lucky circumstance of it carrying three distinct translations of a decree, it enabled the inference of the meaning of hieroglyphs in the early 19th century (Solé, Valbelle, and Rendall 2002).

For digital information, the problem of preserving the knowledge of its intended meaning, its data format, is a lot more complex. We do not have a small set of languages like hieroglyphic Egyptian with distinguishable symbols in use, but rather a variety of different data formats on binary data. Each of them defines the meaning of bits and bytes essentially depending on context, so for accessing contained information, establishing the meaning of data from context needs processing. Yet for this processing, we depend on implementations that are expensive to create, do age over time and become obsolete due to rapid technological change.

Research Problem

Our central research problem is that the current state of specifying data format knowledge is based on semi formal, textual specifications. As these documents are intended for human engineers, application of this knowledge to a problem inevitably depends on human labour, needed for developing suited implementations for a specific technological environment and purpose.

Now, rapid technological change of environments (e.g. hardware, operating systems, programming languages) combined with a variety of processing purposes (e.g. reading, writing, validating, repairing, optimizing) and the ongoing development of data formats constantly retriggers the need for a new development cycle. Complicating matters, reuse is often severely limited, as adaptation of existing source code can be next to impossible due to radical differences in suited implementations. Taking X.509 security certificates as example, developing software can result in widely different implementations for writing them on a Java mobile phone, for reading them in a batch using C++ on a Linux server or for validating them using Assembler on an memory-constrained embedded system.

Developing format-compliant implementations is a highly complex task, yet at the same time, human engineers have cognitive limits and make mistakes. The cost for developing an implementation, e.g. for sufficiently qualified labour, puts economic limits to feasibility for both public institutions and private companies.

Regarding public institutions, current Digital Preservation practices such as evaluating the risk of data format obsolescence in regular intervals and planning for timely data migration tell of this problem. For private companies, there must be a commercial incentive for the development and maintenance of products in support of a specific data format - the monetary value of information contained must match the cost associated with its implementation and support in practice. If the monetary value does not match its cultural or scientific value on a short timescale, products are discontinued or not developed, resulting in a loss of required processing means, the underlying data format knowledge and thus ultimately of access to contained information.

Contribution

For Digital Preservation of information in arbitrary data formats, the current practice of semi-formal, textual specifications and the subsequently required human engineering effort is too expensive to guarantee long term access to information, not speaking about other usual problems such as format-compliance of implementations and authenticity of data.

We therefore propose the formal description of data formats in order to make data format knowledge machine-processable in the first place and thus enable its automated application in a scalable manner, e.g. for extracting information from formatted data or for generating skeleton source code for implementations.

Towards that purpose, we recently published the concept of *Bitstream Segment Graphs* (BSGs) for describing the composition of data (Hartle et al. 2008a). In this paper, we build upon BSGs and contribute a logic-based approach for formal data format specification.

Related Work

Data formats are not only a subject in Digital Preservation, but rather a cross-cutting concern that appears in other disciplines of research as well:

- In *Multimedia*, motivations for research on data formats were the need to specify data formats for MPEG-4, e.g. for Part 2 (Visual) (ISO 2004) on the one hand and the *Universal Multimedia Access* (UMA) vision (Vetro, Christopoulos, and Ebrahimi 2003) in the context of MPEG-21 (ISO 2007) on the other hand, part of which focuses on content adaptation and filtering. The former led to MSDL-S (Eleftheriadis 1996) and its successor Flavor/XFlavor (Eleftheriadis and Hong 2004), whereas the latter resulted in BSDL (ISO 2008). In this domain, contributions in literature are basically restricted to high-level descriptions of bitstreams.
- Regarding *Telecommunication*, the main motivation was the need to specify an efficient representation of a data model in an interoperable manner. This has lead to the Abstract Syntax Notation One (ASN.1) (ITU-T 1997), the generic Encoding Control Notation (ITU-T 2002b) and specific standard encodings such as CER or DER (ITU-T 2002a). For arbitrary data formats that do not fit into these encodings, universal applicability is sometimes claimed for the combination of ASN.1 & ECN, yet such a claim has neither been proven nor substantiated for these two highly complex specifications.

Other disciplines also touch upon the subject of data formats, e.g. the Semantic Web with the problem of making information accessible to machine reasoning, or IT Security with the problem of testing application robustness by the introduction of data errors, so-called *fuzzing* (Miller, Fredriksen, and So 1989)

Approach

In general, we assume a data format to define a lossless digital representation of some structured information for purposes of storage and transmission. A data format therefore defines a set of finite, consecutive bit sequences and a set of structured information. Both sets may be infinite in size and have a one-to-one correspondence.

We thus assume that there exists a bijective mapping function between both sets (for *parsing* and *serialisation*) as well as functions for deciding the membership in either set. For practicability, we require that all three problems (bijective mapping as well as membership in either set) are computable and decidable, that is, there exists a Turing machine that always computes an answer to the problem and halts.

Computational Complexity

Bijectivity of the mapping function does not limit its computational complexity, as it was shown that every single-tape Turing machine can be converted into a logically reversible 3-tape Turing machine (Bennett 1973). Moreover, no general formalism can exist that exactly covers the set of decidable problems, as follows from the *Halting Problem* (Hopcroft and Ullman 1979). Therefore, describing arbitrary data formats requires a formalism which is equal to the Turing machine in computational power. Such a formalism inherits the Halting Problem and thus cannot guarantee decidability by itself.

Decomposing the problem

In order to decompose the problem of formal data format specification, we define a *data format instance* as the bijective mapping between a pair of elements from both sets. We further define a *data format* as a potentially infinite set of data format instances, with the definition intentionally being analogous to that of a formal language (Mateescu and Salomaa 1997).

We therefore decompose the problem of formal data format specification into the problem of describing arbitrary data format instances and the problem of describing a possibly infinite set of bijective mappings.

Model

For the first problem, we have recently proposed a model for describing arbitrary data format instances using the *Bitstream Segment Graph* (Hartle et al. 2008a), which has also been applied for describing exploits in IT Security (Hartle et al. 2008b). For the latter problem, we build upon the BSG model and propose a logic-based approach through fixed-point deduction of BSG instances.

Describing arbitrary data format instances

An abbreviated introduction into Bitstream Segment Graphs is given in this subsection. For a more formalized description, the reader is kindly referred to (Hartle et al 2008a).

Entities

A *bitstream segment* is a finite, consecutive bit sequence such as `01000001`. A *bitstream source* is a defined bitstream segment that is to be described and which follows a certain data format, e.g. a specific image file or a network packet.

A *bitstream transformation* is a bijective mapping of input bitstream segments to output bitstream segments, limited to one of the following normalisations:

Bitstream segment type	Used in encoding?	Used in transformation?		Coverage
Generic	no	no	(as input)	0
Primitive	yes	no	(as input)	1
Structure	no	segmentation	(as input)	length-weighted coverage of successors
Transcode	no	transformation	(as input)	coverage of successor
Fragment	no	concatenation	(as input)	coverage of successor
Composite	no	concatenation	(as output)	coverage of successor

Table 1: Bitstream segment types.

- the *segmentation transformation* that splits one input bitstream segment into two or more ordered output bitstream segments (1:n),
- a class of *block transformations* which transform one input bitstream segment into one output bitstream segment (1:1), and
- the *concatenation transformation* that joins two or more ordered input bitstream segments into one output bitstream segment (n:1).

Examples for these normalised bitstream transformations are the segmentation of a data structure into its fields, block transformations such as GZIP compression, AES encryption or Reed-Solomon error-correction, or the aggregation of a fragmented multimedia stream in an Apple QuickTime container. Arbitrary (n:m) bitstream transformations can be constructed through sequential composition of multiple normalised transformations. A bitstream transformation connects input and output segments as *predecessors* and *successors*, respectively. No cycles may be formed through bitstream transformations either directly or indirectly.

A bitstream encoding is a bijective mapping between a bitstream segment and a typed literal, representing some information. For example, the bit sequence `01000001` represents the number 65 for a big-endian unsigned integer encoding, whereas for an ASCII encoding, it represents the letter `A`.

Every bitstream segment belongs to one of 6 bitstream segment types, depending upon its participation in bitstream transformations and bitstream encodings as listed in Table 1. For example, it may be a *structure* composed from two or more successor bitstream segments, a *primitive* if it represents an encoded literal, or a *generic* if it does not participate in a bitstream transformation or a bitstream encoding.

The *coverage* of a bitstream segment is a measure in the range between 0 and 1 and expresses how completely a bitstream segment is mapped to encoded literals through its successor(s). It is computed depending on the bitstream segment type (see Table 1). For example, for a structure bitstream segment `a` with two primitive segments as successors, the coverage of `a` would be 1. In case of one primitive segment and a generic segment of equal length as successors, the coverage of `a` would be 0.5. The coverage of a BSG instance refers to that of its bitstream source.

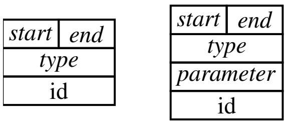

Figure 1: Visual representations: generic, structure and composite bitstream segments (left); fragment, primitive and transcode bitstream segments (right).

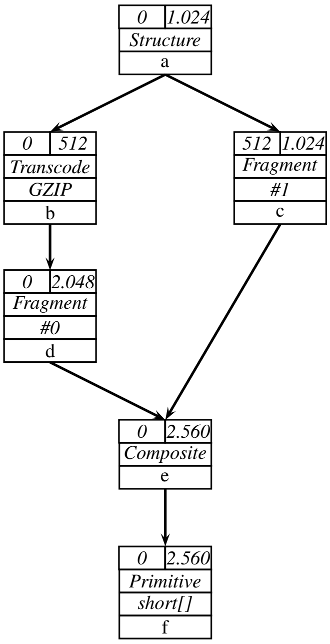

Figure 2: Minimal example of a BSG instance.

A *Bitstream Segment Graph* (BSG) is now a rooted, acyclic graph that is defined from a bitstream source, a set of bitstream transformations and a set of bitstream encodings, where the nodes correspond to bitstream segments and the edges to transformations. It describes the composition of a bitstream source from primitive bitstream segment(s). For a visual representation of a BSG instance, bitstream segments are depicted as in Figure 1.

Properties

A bitstream segment `x` has a set of namespaced *properties*, denoted as `ns:property(x,`v_0`,…,`v_n`)`. For the BSG model, this includes placement information such as an inclusive `bsg:start` position, a `bsg:length` and an exclusive `bsg:end` position, all measured in bits and relative to the context provided by its predecessors. For

Predicate	Behaviour
`math:lt(?a,?b)`	Tests the formula `?a` < `?b`.
`math:lte(?a,?b)`	Tests the formula `?a` <= `?b`.
`math:eq(?a,?b)`	Tests the formula `?a` = `?b`.
`math:product(?a,?b,?c)`	Computes the formula `?a` * `?b` = `?c` if two parameters are ground and no division by zero occurs, and assigns the result to the third variable parameter. Tests the formula if all parameters are ground.
`math:sum(?a,?b,?c)`	Computes the formula `?a` + `?b` = `?c` if two parameters are ground and assigns the result to the third variable parameter. Tests the formula if all parameters are ground.
`util:concat(?a,?b,?c)`	Concatenates ground strings `?a` and `?b` and binds the result to variable `?c`. Tests whether the concatenation of `?a` and `?b` corresponds to `?c` if all parameters are ground.
`util:sourceLength(?a,?b)`	Gets the length in bits of the ground file reference `?a` and binds it to variable `?b`. Tests whether file reference `?a` has length `?b` in bits if both are ground.
`util:skolem(?a,…,?c)`	*Skolem function provided for existential quantification*. Maps the set of ground parameters (`?a`, …) to a value and binds it to variable `?c`. Maps a ground `?c` to a set of values and binds them to variables (`?a`, …). Tests whether (`?a`,…) and `?c` map to each other if all parameters are ground.
`util:value(?a,?b)`	Decodes the contained literal of a ground primitive bitstream segment `?a` if it is bsg:resolved, and assigns the result to variable `?b`. Tests whether the bitstream segment `?a` contains the literal `?b` if both parameters are ground.

Table 2: List of computable predicates.

example, the first successor segment of a structure segment starts at bit 0. Further properties include the bitstream segment `bsg:type`, one or more `bsg:semantics` as identifiers or a `bsg:codec` identifier for transcode bitstream segments. For example, for the segments `b` and `c` in Figure 2, we can state properties such as `bsg:start(b,0)`, `bsg:length(c,512)` or `bsg:codec(b,GZIP)`.

Relations

Between any two bitstream segments `x` and `y`, namespaced *relations* may exist, denoted as `ns:relation(x,y,`v_0`,…,`v_n`)`. For the BSG model, this includes neighbourship relations between bitstream segments in a structure bitstream segment as `bsg:leads` and `bsg:follows`, and composition relations such as `bsg:successor` and `bsg:predecessor` with `bsg:firstSuccessor` and `bsg:lastSuccessor` as special cases. For example, for the segments `a`, `b` and `c` in Figure 2, we can state relations such as `bsg:firstSuccessor(a,b)`, `bsg:leads(b,c)` and `bsg:predecessor(c,a)`.

Using suited types of bitstream transformations and encodings, the composition of arbitrary data format instances can be described using BSG instances. Besides the visual representation, we can represent a BSG instance through facts regarding BSG-related properties and relations.

Describing possibly infinite sets of data format instances

We define a potentially infinite set of bijective data format instances through the set of stable models resulting from a set of first-order logic rules, expressed as implications or biconditionals. For rules, predicates are used that refer to either deduced or computed facts. In terms of existing logic languages, it resembles Datalog (Ullman 1989) extended with functions.

Deducible predicates refer to facts that were either given initially or subsequently deduced through rules. They are not limited to BSG-related properties and relations only, but may also include predicates for intermittent facts which may be needed for deducing a BSG instance. For deduced predicates, the open world assumption applies, as a currently unknown fact may become known later. *Computable predicates* refer to facts that can be computed directly (see Table 2). They handle aspects such as decoding the literal `?l` of a primitive bitstream segment `?x` from the so-far deduced, partial BSG instance through `bsg:value(?x,?l)`, or for solving the equation `?v=?u+1` through `math:sum(?u,1,?v)` if either `?u` or `?v` are known. These predicates can choose between the open world assumption and the closed world assumption, as they can decide to refute facts that will always fail, such as `math:sum(1,2,4)`.

Predicates have parameters that can either be *ground* and thus have a specific value, or be a *variable*. A *mode* of a predicate declares for each of its parameters whether it is ground or variable. Computable predicate may support arbitrary modes, e.g. allowing `math:sum` to compute `math:sum(?u,4,5)` as well as `math:sum(1,?v,5)` and `math:sum(1,4,?w)`, or test `math:sum(1,4,5)`.

Using these types of predicates, we can build rules as implications or biconditionals. These rules can be partitioned into model-specific rules that capture properties and relations inherited from the BSG model itself, and format-specific rules that represent data format knowledge. For example, a BSG-specific rule is that two

#	Rule
M1	`bsg:source(?a,?f) & util:sourceLength(?f,?l) → bsg:start(?a,0) & bsg:length(?a,?l)`
M2	`bsg:length(?a,?l) & bsg:end(?a,?e) & math:sum(?s,?l,?e) → bsg:start(?a,?s)`
M3	`bsg:start(?a,?s) & bsg:end(?a,?e) & math:sum(?s,?l,?e) → bsg:length(?a,?l)`
M4	`bsg:start(?a,?s) & bsg:length(?a,?l) & math:sum(?s,?l,?e) → bsg:end(?a,?e)`
M5	`bsg:start(?a,?s) & bsg:length(?a,?l) & bsg:end(?a,?e) → math:sum(?s,?l,?e)`
M6	`bsg:leads(?a,?b) ↔ bsg:follows(?b,?a)`
M7	`bsg:leads(?a,?b) & bsg:end(?a,?p) ↔ bsg:follows(?b,?a) & bsg:start(?b,?p)`
M8	`bsg:firstSuccessor(?a,?b) → bsg:successor(?a,?b)`
M9	`bsg:lastSuccessor(?a,?b) → bsg:successor(?a,?b)`
M10	`bsg:successor(?a,?b) → bsg:predecessor(?b,?a)`
M11	`bsg:successor(?a,?b) & bsg:leads(?b,?c) → bsg:successor(?a,?c)`
M12	`bsg:successor(?a,?b) & bsg:follows(?b,?c) → bsg:successor(?a,?c)`
M13	`bsg:firstSuccessor(?a,?b) → bsg:start(?b,0)`
M14	`bsg:lastSuccessor(?a,?b) & bsg:length(?a,?c) → bsg:end(?b,?c)`
M15	`bsg:lastSuccessor(?a,?b) & bsg:end(?b,?c) → bsg:length(?a,?c)`
M16	`bsg:start(?a,?s) & bsg:length(?a,?l) & bsg:end(?a,?e) & bsg:type(?a,?t) & bsg:source(?a,?f) → bsg:resolved(?a)`
M17	`bsg:successor(?a,?b) & bsg:start(?b,?s) & bsg:type(?b,?t) & bsg:resolved(?a) → bsg:resolved(?b)`

Table 3: List of model-specific rules.

neighbouring bitstream segments `b` and `c` share a boundary, so from the facts `bsg:follows(b,c)` and `bsg:end(b,512)`, the fact `bsg:start(c,512)` can be concluded. From the data format instance in Figure 2, we could assume as format-specific rule that from the facts `bsg:source(a,…)` and `bsg:firstSuccessor(a,b)`, the fact `bsg:type(b,'bsg:transcode')` follows.

For deducing a BSG instance, initial knowledge on a specific bitstream source is given, such as the fact `bsg:source(a, 'oi2n0g16.png')`. Through a series of iterative steps, the set of rules is applied in a monotone deduction process. In each step for every rule, it is tried to match the antecedents with previously deduced knowledge. If the antecedent of a rule matches, then for its conclusion, the computable predicates are tested and the deducible predicates are deduced. Should a computable predicate fail in this test, the reasoning process aborts, as a conclusion does not hold. This allows the use of validation rules that assert certain properties, e.g. that for all bitstream segments, its respective `bsg:start` and `bsg:length` have to sum up to its `bsg:end`, which can be violated in case of contradictory information contained in a damaged or erroneous bitstream source. When no new facts are deduced in a step, then a fixed point consisting of the deducible facts of a BSG instance is reached.

If a fixed point is reached, the resulting BSG facts can then be translated into a BSG instance for that bitstream source. This requires post-processing steps such as assigning the generic bitstream segment type whenever no type was deduced for a bitstream segment. The deduction of a BSG instance therefore can either

- abort with a computable predicate refuting a fact in a rule conclusion, indicating that a conclusion does not hold and thus the bitstream source does not conform to the specified data format,
- reach a fixed point with a coverage x < 1, indicating that there are bitstream segments in this data format instance not specified in the set of rules, or
- reach a fixed point with a coverage x = 1, indicating that this data format instance is completely covered by the set of rules.

Building a set of rules as data format knowledge is typically an incremental process. It starts with the collection of bitstream sources for a corpus that represents a specific format, and the definition of an initial set of rules. This set of rules can be improved step-by-step by computing the BSG instance for every bitstream source in the corpus and computing its coverage. One then can select BSG instances with a coverage x < 1 and focus on generic bitstream segments which need to be described further through additional rules. Actual knowledge on how these generic bitstream segments are actually composed may come from consulting textual specifications, existing implementations or through try-and-error reverse engineering efforts. Repeating this process increases the overall coverage of BSG instances in the corpus. For a corpus, a *fitting* set of rules is found if the coverage reaches 1 for all of its BSG instances.

Evaluation

In order to apply our approach, we implemented a reasoning system in Java, defined suited interfaces for processing bitstream transformations and bitstream encodings, and implemented components for handling certain transformations and encodings as required.

Setup

For evaluation, we decided to describe a small subset of the Portable Network Graphics (PNG) image format. We required that of this subset, some data format instances

#	Rule
F1	bsg:source(?a,?f) → bsg:semantics(?a,'png:root')
F2	bsg:semantics(?r,'png:root') → util:skolem('F2',?r,?s) & bsg:type(?r,'bsg:structure') & bsg:firstSuccessor(?r,?s) & bsg:semantics(?s,'png:signature')
F3	bsg:semantics(?s,'png:signature') → util:skolem('F3',?s,?f) & bsg:leads(?s,?f) & bsg:semantics(?f,'png:chunk')
F4	bsg:semantics(?c,'png:chunk') → util:skolem('F3',?c,?l) & bsg:firstSuccessor(?c,'png:chunk') & bsg:semantics(?l,'png:chunk-length')
F5	bsg:semantics(?l,'png:chunk-length') → util:skolem('F5',?l,?t) & bsg:leads(?l,?t) & bsg:semantics(?t,'png:chunk-type')
F6	bsg:semantics(?l,'png:chunk-length') & bsg:value(?l,0) & bsg:leads(?l, ?t) & bsg:successor(?ch,?l) → util:skolem('F6',?l,?t,?ch,?cr) & bsg:lastSuccessor(?ch,?cr) & bsg:leads(?t,?cr) & bsg:semantics(?cr,'png:chunk-crc')
F7	bsg:semantics(?l,'png:chunk-length') & bsg:value(?l,?v) & math:lt(0,?v) & bsg:leads(?l,?t) & bsg:successor(?ch,?l) & math:product(?v,8,?lv) → bsg:leads(?t,?d) & bsg:leads(?d,?cr) & bsg:lastSuccessor(?ch,?cr) & bsg:length(?d,?lv) & bsg:semantics(?d,'png:chunk-data') & bsg:semantics(?cr,'png:chunk-crc')
F8	bsg:semantics(?t,'png:signature') → bsg:type(?t,'bsg:primitive') & bsg:encoding(?t,'http://www.dataformats.net/2008/04/bsg-encodings#ascii-string') & bsg:length(?t,64)
F9	bsg:semantics(?t,'png:chunk-length') → bsg:type(?t,'bsg:primitive') & bsg:encoding(?t,'http://www.dataformats.net/2008/04/bsg-encodings#msbf-uint') & bsg:length(?t,32)
F10	bsg:semantics(?t,'png:chunk-type') → bsg:type(?t,'bsg:primitive') & bsg:encoding(?t,'http://www.dataformats.net/2008/04/bsg-encodings#ascii-string') & bsg:length(?t,32)
F11	bsg:semantics(?t,'png:chunk-crc') → bsg:type(?t,'bsg:primitive') & bsg:encoding(?t,'http://www.dataformats.net/2008/04/bsg-encodings#msbf-uint') & bsg:length(?t,32)
F12	bsg:successor(?ch,?t) & bsg:semantics(?ch,'png:chunk') & bsg:semantics(?t,'png:chunk-type') & bsg:value(?t,?v) → util:concat('png:chunk:',?v,?ct) & bsg:semantics(?ch,?ct)
F13	bsg:successor(?r,?c) & bsg:semantics(?c,'png:chunk') & bsg:end(?c,?ce) & bsg:length(?r,?rl) & math:lt(?ce,?rl) → util:skolem('F13',?c,?ce,?r,?rl,?nc) & bsg:leads(?c,?nc) & bsg:semantics(?nc,'png:chunk')
F14	bsg:semantics(?r,?c) & bsg:semantics(?c,'png:chunk') & bsg:end(?c,?ce) & bsg:length(?r,?rl) & math:eq(?ce,?rl) → bsg:lastSuccessor(?r,?c)

Table 4: Excerpt of format-specific rules for a limited PNG subset. Due to length considerations, this list is limited to describing a PNG image down to the level of chunk structures.

should at least be sufficiently complex as to require all three types of normalised bitstream transformations (segmentation transformation, block transformation and concatenating transformation) from the BSG model.

We found a suited subset of PNG images, namely those where compressed image data is stored as separate fragments in so-called IDAT chunks. For building a suited corpus, we examined the PNG Test Suite (van Schaik 1998) with 156 PNG images for compliance testing, including corrupted files and extreme variants, and selected 8 images with filename pattern `oi??????.png`.

Regarding the granularity of description, we allowed primitive bitstream segments to represent arrays of encoded literals. Without this consideration, the decomposition of arrays such as pixel data into individual pixels would have bloated the resulting description of a data format instance without substantial benefit.

We built a fitting set of rules for our corpus, consisting of 17 model-specific rules (see Table 3) and 36 format-specific rules (see Table 4 for an excerpt).

Data format rules

Regarding model-specific rules, we start with rules on placement regarding a bitstream segment. This begins with a rule for deducing `bsg:start` and `bsg:length` from an initially given `bsg:source` (M1). If any two of `bsg:start`, `bsg:length` and `bsg:end` are given for a bitstream segment, the remaining fact can be deduced (M2-M4). Moreover, if all facts are given for a bitstream segment, it can be validated for ensuring consistency (M5). Further rules include aspects of bitstream segments in structures, such as neighbourship (M6, M7), successorship (M8-M12), placement (M13-M15) and resolvability (M16, M17), whereas the latter is necessary for decoding the contained literal of primitive bitstream segments.

Finally, we come to format-specific rules on our PNG subset. We start with a rule that deduces the PNG-specific type of 'png:root' for a bitstream source (F1). For such a bitstream segment, we can deduce that there exists a first successor bitstream segment `?s` with

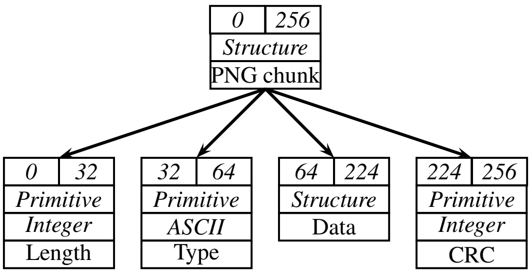

Figure 3: BSG instance for a PNG chunk.

`bsg:semantics(?s,'png:signature')` (F2). For a 'png:signature', there exists a following 'png:chunk' structure (F3) as shown in Figure 3, which again always begins with a 'png:chunk-length' bitstream segment (F4), followed by a 'png:chunk-type' bitstream segment (F5).

If the value of a 'png:chunk-length' is 0, then the 'png:chunk-type' is followed directly by the 'png:chunk-crc' bitstream segment as last successor of the chunk (F6). Otherwise, the 'png:chunk-type' bitstream segment is followed by a variable-length 'png:chunk-data' bitstream segment and again the 'png:chunk-crc' bitstream segment (F7). Details on bitstream segments such as their type, encoding and length are provided for 'png:signature' (F8), 'png:chunk-length' (F9), 'png:chunk-type' (F10) and 'png:chunk-crc' (F11) bitstream segments. The PNG-specific type of the chunk is deduced from the 'png:chunk-type' value and assigned as `bsg:semantics` to the chunk (F12). The remaining rules listed in Table 4 state that if there is space left after a chunk, there exists another one following (F13), otherwise the chunk is the last successor of the bitstream source (F14). Further rules handle chunk-specific aspects, e.g. for the IHDR chunk which contains information on image width and height.

Example deduction steps

For a given initial fact

```
bsg:source('root','oi2n0g16.png')
```

the deduction process tries to apply all rules to deduce new facts. In the first step, only the rules F1 and M1 are applicable, which yield the following new facts:

```
bsg:semantics('root','png:root') &
bsg:start('root',0) &
bsg:length('root',1432)
```

Again, the deduction process tries to apply all rules, this time on an increased set of facts. In step 2, the rules F2 and M4 yield the following:

```
bsg:type('root','bsg:structure') &
bsg:firstSuccessor('root','sc1') &
bsg:semantics('sc1','png:signature') &
bsg:end('root',1432)
```

The process of deduction is repeated until either no new facts can be deduced, or a computable predicate refutes a fact in a conclusion. The resulting facts from the reached fixed point describe a BSG instance for the PNG image oi2n0g16.png, which is part of the corpus and has a coverage of 1.0.

Result

After building a fitting set of rules with coverage of 1.0 for our corpus, we tested the set on all remaining PNG images from the PNG Test Suite. We obtained a coverage of 1.0 for 64 images, with the remaining 89 valid images having an average coverage of 0.79. Three corrupt images belonging to the test suite were excluded from the evaluation, as the fitting set of rules did not contain verifying rules for PNG-specific properties.

For a fitting set of rules over the entire PNG Test Suite, additional rules need to be included for palette handling (PLTE and sPLT chunks), transparency (tRNS chunk), background colour (bKGD chunk), textual data (tEXt and zTXt chunks) and other aspects. To estimate the effect of adding further rules, we added two preliminary rules for handling PLTE chunks and re-evaluated our rules on the corpus. We obtained a coverage of 1.0 for 78 images, with the remaining 75 valid images having an average coverage of 0.91.

During evaluation, the deduction process computed a fixed point and halted on all instances. Since errors may be present in a set of rules preventing a fixed point to be reached, a primitive approach on handling the Halting Problem is to place a limit on the iteration steps and abort the deduction beyond that limit. We discovered that the typical number of iterative steps required for our set of rules to reach a fixed point on valid PNG images ranges from 72 up to 170 steps. In case of the image file oi9n2c16.png, more than 3,000 iteration steps were required, as compressed image data is present as fragments with a length of 8 bit, each encapsulated into a separate IDAT chunk. This can be considered an extreme example, but demonstrates what is still considered legal in terms of the original specification. Since data format instances of other data formats such as Apple QuickTime movies have a more complex structure which requires an even higher number of iterations, the use of a semi-naive evaluation method for the deduction process as known from Datalog (Ullman 1989) is absolutely essential.

Discussion

The set of rules we tested is quite small, yet describes central elements of PNG files. 'Unexplained' bitstream segments can be readily identified due to the generic bitstream segment type and the coverage measure, and thus allow for incremental development of data format rules. Testing this approach, incrementally adding rules for PLTE chunks to describe palette information had been quite simple and resulted in a significant increase regarding the coverage of nearly all images in the PNG Test Suite.

Regarding data formats in general, our approach maps the diversity of data formats to format-specific data format rules, bitstream transcodings and bitstream encodings. We assume that some bitstream transcodings and a majority of bitstream encodings may be shared among multiple data formats. For example, PNG employs a scanline transformation to increase the efficiency of a subsequent GZIP compression transformation; the GZIP compression is likely to be reusable, whereas the scanline transformation is highly PNG-specific. The bitstream encodings we encountered so far are basically the ASCII encoding used for PNG chunk types and a bit-endian unsigned integer encoding used for numerical values, which are easily reusable, e.g. in the context of Apple QuickTime.

The set of rules includes model-specific rules that validate the consistency of essential model-specific properties. Due to the complexity of PNG, adding rules for validating all PNG-specific properties is nontrivial and requires specifically corrupted image files for testing the corresponding rules. Our tested set of rules is over-accepting in terms of a formal language when compared to the PNG specification.

We decided to use first-order predicate logic in our approach, yet it may be possible that data formats have rules which are more naturally expressed using fragments of higher-order logics, e.g. when having to express rules on sets of segments. For example, when multiple IDAT chunks are present in an BSG instance, these have to be concatenated in order of their appearance, yet formulating the corresponding rules was non-intuitive. We assume that complex data format rules will at times translate into specialised computable predicates and require larger, more complex sets of rules.

Summary and Conclusion

We have presented an approach for describing arbitrary data formats as a possibly infinite set of data format instances, building upon the Bitstream Segment Graph model. In contrast to previous related work, we can describe arbitrary data format instances down to contained primitives even when real-life aspects such as compression or fragmentation are present. We applied our approach to the description of a sufficiently complex subset of the PNG image format and were able to show that a quite small number of rules is capable of describing a significant part of PNG images. Furthermore, our approach allows the measurement on how completely a set of rules describes a data format instance, which supports the incremental development of data format rules over time.

It therefore provides some means for formal specification of data formats, which may be of use for the specification of new data formats and for the documentation of existing ones. This can especially be helpful for data formats which are undisclosed or which are deviations. For Digital Preservation, a formal data format specification may provide for ``a last line of defense" by allowing to extract contained information if a fitting set of rules exists.

Acknowledgements

The authors would like to thank Gina Häußge for feedback, comments and corrections on various drafts of the paper.

References

Bennett, C. H. 1973. Logical Reversibility of Computation. *IBM Journal of Research and Development* 17(2):525–532.

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*, 816–819. New York, NY, USA: ACM Press.

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

Hartle, M.; Möller, F.-D.; Travar, S.; Kröger, B.; and Mühlhäuser, M. 2008a. Using Bitstream Segment Graphs for Complete Data Format Instance Description. In *Proceedings of the Third International Conference on Software and Data Technologies (ICSOFT).*

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

Hopcroft, J. E., and Ullman, J. D. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley.

ISO. 2004. ISO/IEC 14496-2:2004: Information technology – Coding of audio-visual objects – Part 2: Visual. ISO, Geneva.

ISO. 2007. ISO/IEC 21000-7:2007: Information technology – Multimedia framework (MPEG-21) – Part 7: Digital Item Adaptation. ISO, Geneva.

ISO. 2008. ISO/IEC 23001-5:2008: Information technology – MPEG systems technologies – Part 5: Bitstream Syntax Description Language (BSDL). ISO, Geneva.

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

ITU-T. 2002a. Recommendation X.690 (07/02) — ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER). ITU-T, Geneva.

ITU-T. 2002b. Recommendation X.692 (03/02) — ASN.1 Encoding Rules: Specification of Encoding Control Notation (ECN). ITU-T, Geneva.

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

Miller, B. P.; Fredriksen, L.; and So, B. 1989. An Empirical Study of the Reliability of UNIX Utilities. Technical report.

Solé, R.; Valbelle, D.; and Rendall, S. 2002. *The Rosetta Stone: The Story of the Decoding of Hieroglyphics*. Four Walls Eight Windows.

Ullman, J. D. 1989. *Principles of Database and Knowledge-Base Systems, Volume II*. Computer Science Press.

van Schaik, W. 1998. PngSuite - The Official Set of PNG Test Images. Available online at http://www.schaik.com/pngsuite/pngsuite.html, last accessed 2008-08-015.

Vetro, A.; Christopoulos, C.; and Ebrahimi, T. 2003. From the guest editors: Universal Media Access. *IEEE Signal Processing Magazine* 20(2):16–16.