A Formal, Declarative Approach to Data Format Description

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

Dissertation

zur Erlangung des akademischen Grades Dr.-Ing.

vorgelegt von
Dipl.-Inform. (FH) Michael Hartle
geboren in Frankfurt am Main

Tag der Einreichung:	7. Juni 2010
Tag der Disputation:	16. Juli 2010
Referenten:	Prof. Dr. Max Mühlhäuser, Darmstadt
	Prof. Dr. Andreas Rauber, Wien

Darmstadt 2010
D17

Ehrenwörtliche Erklärung [1]

Hiermit erkläre ich, die vorgelegte Arbeit zur Erlangung des akademischen Grades "Dr.-Ing." mit dem Titel "A Formal, Declarative Approach to Data Format Description" selbständig und ausschließlich unter Verwendung der angegebenen Hilfsmittel erstellt zu haben. Ich habe bisher noch keinen Promotionsversuch unternommen.

Darmstadt, den 7. Juni 2010

Dipl.-Inform. (FH) Michael Hartle

[1]Gemäß §9 Abs. 1 der Promotionsordnung der TU Darmstadt

Wissenschaftlicher Werdegang des Verfassers[2]

09/1998–03/2003	Studium der Informatik an der Fachhochschule Darmstadt Abschluss mit Note 1,3
09/2002–03/2003	Diplomarbeit an der Fachhochschule Darmstadt Titel “Real-Time Generation of Planetary Landscapes”
07/2004–07/2010	Wissenschaftlicher Mitarbeiter am Fachgebiet Telekooperation (Prof. Dr. Mühlhäuser) an der Technischen Universität Darmstadt
07/2004–07/2010	Technischer Mitarbeiter am e-learning center im Hochschulrechenzentrum der Technischen Universität Darmstadt
09/2006–02/2010	Assoziiertes Mitglied im Graduiertenkolleg 1223 “Qualitätsverbesserung im E-Learning durch rückgekoppelte Prozesse”

[2]Gemäß §20 Abs. 3 der Promotionsordnung der TU Darmstadt

Abstract

The concept of data formats is central to information storage and exchange, as it coins the process of how information is written to and read back from format-compliant data by senders and receivers. In contrast to the widespread use of natural-language descriptions intended for human engineers, and of procedural definitions of format-compliant components, describing data format knowledge in a formal, declarative manner is necessary for making this knowledge machine-processible, enabling its flexible, automated application to format-compliant data. To that effect, data format knowledge is considered both on the level of format-compliant data as a data format instance and on the level of a data format consisting of such instances.

In a survey of current State of the Art in Data Format Description, examined related work from the data-centric research domains of Digital Preservation, Multimedia and Telecommunication show a lack suitable formalised models for universal applicability. As well, examined related work provides only a subset of the four necessary descriptive capabilities to describe data which may be primitive, structured, transcoded or fragmented.

In the analysis, a formalisation is presented which is based on the research hypothesis that a data format defines a normative set of lossless information representations, where there exists a bijective mapping between interal representations of senders / receivers, and external representations that are exchanged as format-compliant data. The formalisation is universally applicable for arbitrary data formats, is suitable for both so-called lossless and lossy data formats, and leads to the notion of four elementary descriptive capabilities, which exactly match those used in the State of the Art survey. A valid Portable Network Graphics (PNG) raster image is given as "litmus test" for data format description, as its description exercises all four elementary descriptive capabilities. Based on the formalisation, it is shown that a universal approach to data format description is too powerful in computational terms as to be able to guarantee termination, that the tractability of bijective mapping functions and their inverses is neither given nor necessarily related, and that one-to-one correspondence of a bijective mapping function can be guaranteed using information-preserving, Turing-complete Reversible Turing Machines.

Building on the formalisation given in the analysis, the thesis defines the Bitstream Segment Graph (BSG) model for describing arbitrary data format instances. For BSG instances, representations are defined both for visualisation as well as for storage and exchange through machine-processible, RDF-based representations. Incremental construction and modification of BSG instances is enabled through a closed set of operations, and the "coverage" of a BSG instance is defined as a measure of its completeness. Actual tool support for the construction, modification and exploration of BSG instances on arbitrary data is provided through the Apeiron BSG Editor. Applications of the BSG model are demonstrated through the description of the PNG image "litmus test" from the previous analysis, and for the description of an exploit in the context of IT Security.

Building on the BSG model, the thesis defines the BSG Reasoning approach for describing arbitrary data formats as potentially infinite sets of data format instances. Using logic rules, a BSG instance can be inferred for a given bit sequence which is considered to be format-compliant data. The BSG Reasoning approach defines the representation of rulesets for storage and exchange. Applications of BSG Reasoning are demonstrated through the description of a PNG image file format subset and through an outlined approach for format-aware fuzzing of bitstreams in IT Security. The PNG image file format subset described through BSG Reasoning exercises all elementary descriptive capabilities previously identified in the analysis, and it is shown that the resulting set of logic rules, despite a low number of format-specific rules, already yields a high coverage of inferred BSG instances on a number of valid PNG images.

The thesis closes with a retrospection, conclusions and an outlook on potential future research on the BSG model and the BSG Reasoning approach, focusing on aspects such as the computer-aided reverse-engineering of data format rules, or the use of reversible programming languages for the definition of lossless coding and transformation functions.

Zusammenfassung

Die Speicherung und der Austausch von Informationen ist eng mit dem Begriff des Datenformats verknüpft. Ein Datenformat legt fest, wie Informationen format-konform von einem Sender als Daten geschrieben und aus diesen von einem Empfänger wieder gelesen werden können. Obwohl natürlich-sprachliche Beschreibungen für menschliche Ingenieure heute häufig genutzt werden, und format-konforme Abläufe teilweise prozedural beschrieben werden, hätte eine formale, deklarative Beschreibung von Datenformat-Wissen den Vorteil, daß dieses ohne Bindung an einen konkreten Ablauf und ohne den Umweg über Menschen maschinen-verarbeitbar ist, und damit flexibel und automatisiert auf format-konforme Daten angewandt werden kann. Im Rahmen dieser Dissertation wird Datenformat-Wissen sowohl auf der Ebene von format-konformen Daten als Datenformat-Instanz als auch auf der Ebene eines Datenformats betrachtet, welches aus Datenformat-Instanzen besteht.

Im Rahmen einer Begutachtung verwandter Arbeiten im Bereich der Datenformat-Beschreibung werden Ansätze in den daten-orientierten Forschungsgebieten der Digitalen Erhaltung, Multimedia und Telekommunikation untersucht, und es wird festgestellt, dass geeignete, formalisierte Modelle fehlen, welche universell für die Beschreibung von Datenformaten anwendbar sind. Darüber hinaus hat sich gezeigt, dass die betrachteten Ansätze nur teilweise die notwendigen beschreibenden Fähigkeiten haben, welche erforderlich sind, um den Aufbau von Daten zu beschreiben, welche primitive Werte enthalten, eine Struktur darstellen, einer Block-Transformation unterzogen wurden oder aber in fragmentierter Form vorliegen.

In einer Analyse wird daher eine Formalisierung des Datenformat-Begriffs entwickelt, welche von der Annahme ausgeht, dass ein Datenformat ein normatives Set von verlustfreien Informations-Repräsentationen darstellt. Für ein solches Set existiert eine bijektive Abbildung zwischen der internen Repräsentation eines Senders / Empfängers und der korrespondierenden externen Repräsentation, welche in Form format-konformer Daten ausgetauscht wird. Diese Formalisierung ist universell für beliebige Datenformate anwendbar, also auch für sogenannte verlustbehaftete und verlustfreie Datenformate, und führt zum Konzept von elementaren beschreibenden Fähigkeiten, welche sich genau mit denen decken, welche in der Begutachtung verwendet wurden. Auf Basis dieser Fähigkeiten wird ein gültiges Bild im Dateiformat Portable Network Graphics (PNG) als "Lackmus-Test" für Ansätze der Datenformat-Beschreibung vorgestellt, da dessen Beschreibung alle vier elementaren beschreibenden Fähigkeiten voraussetzt. Auf Basis der Formalisierung wird dann gezeigt, dass ein universell anwendbarer Ansatz zur Datenformat-Beschreibung zu mächtig ist, als dass dessen Terminierung noch garantiert werden kann. Ferner wird gezeigt, dass bijektive Abbildungsfunktionen und ihre Inversen weder effizient sein müssen, noch dass die Effizienz einer bijetiven Abbildungsfunktion und ihrer Inversen im Zusammenhang stehen müssen. Zu guterletzt wird gezeigt, dass die für eine bijektive Abbildung erforderliche Korrespondenz von internen und externen Repräsentationen dadurch garantiert werden kann, dass man diese über eine informations-erhaltende, Turing-vollständige "Reversible Turing-Maschine" definiert.

Aufbauend auf der Formalisierung der Analyse wird in dieser Dissertation das Bitstream Segment Graph (BSG)-Modell definiert, welches der Beschreibung beliebiger Datenformat-Instanzen dient. Für Instanzen des BSG-Modells sind sowohl visuelle Repräsentationen als auch maschinen-verarbeitbare, RDF-basierte Repräsentation für die Speicherung und den Austausch definiert. Die schrittweise Konstruktion und Modifikation von BSG-Instanzen wird durch ein geschlossenes Set von Operationen ermöglicht, und mittels dem Maß der "Abdeckung" einer BSG-Instanz kann deren Vollständigkeit bestimmt werden. Mithilfe des Apeiron BSG Editor ist die Konstruktion, Modifikation und Betrachtung von BSG-Instanzen auf eigenen Daten in der Praxis möglich. Die Anwendung des BSG-Modells wird demonstriert, indem eine Beschreibung des PNG-Bilds aus dem "Lackmus-Test" der Analyse vorgenommen wird, und indem der Aufbau eines Exploit im Kontext der IT-Sicherheit mittels einer Beschreibung näher erklärt wird.

Aufbauend auf dem BSG-Modell beschreibt diese Dissertation den BSG Reasoning-Ansatz, um beliebige Datenformate als potentiell unendliche Sets von Datenformat-Instanzen zu beschreiben. Mithilfe von Logik-Regeln kann eine BSG-Instanz auf einer gegebenen Bitfolge erschlossen werden, von der initial angenommen wird, dass sie format-konform ist. Dieser Ansatz definiert auch die Repräsentation von Regel-Sets zur Speicherung und zum Austausch. Die Anwendung des BSG Reasoning-Ansatzes wird durch die Beschreibung eines Subsets des PNG-Datenformats demonstriert, sowie durch die Beschreibung eines Ansatzes zum format-spezifischen Fuzzing von Binärdaten im Kontext von IT-Sicherheit ergänzt. Die Beschreibung des PNG-Datenformat-Subsets mittels des BSG Reasoning-Ansatzes nutzt alle vier elementaren beschreibenden Fähigkeiten, welche zuvor in der Analyse identifiziert wurden, und es wurde gezigt, dass das hierfür verwendete Set an Logik-Regeln trotz seines geringen Umfangs bereits in der Lage ist, BSG-Instanzen mit einem hohen Grad an Abdeckung für eine Reihe von gültigen PNG-Bildern zu erschliessen.

Die Dissertation schliesst mit einem Rückblick über die gesamte Arbeit, zieht Schlussfolgerungen und bietet einen Ausblick auf künfige Forschung im Hinblick auf das BSG-Modell und den BSG Reasoning-Ansatz, speziell im Hinblick auf Aspekte wie der maschinell unterstützten Analyse von Daten und den jeweils zugrundeliegenden Datenformat-Regeln, oder aber der Verwendung von reversiblen Programmiersprachen zur Definition von Kodierungs- und Transformationsfunktionen.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The concept of data formats is central to information storage and exchange. A data format defines how information is represented digitally as bits, bytes or characters, forming higher-level data structures. It therefore coins the process of *how to determine syntax and semantics of data* in order to access the information represented by it, and to process it in a meaningful manner.

To pass digitally represented information between a sender and a receiver, an agreement is needed on the semantics of data to be transmitted, and on the data format to be used. The agreement is used by a sender to determine how to encode and serialise information to be sent into a sequence of bits, which is then passed on as a message. The actual composition of format-compliant data forming the message depends on the data format agreed for the exchange, and is thus only given implicitly. Therefore, the agreement is also used by a receiver of the message, where the bit sequence is parsed and decoded, enabling access to represented information for further processing. In this process, data format knowledge is applied to make the actual composition of format-compliant data explicit again.

It is therefore necessary that some representation of the data format is given and shared between these parties as part of the agreement, leading to the problem of describing data format knowledge.

1.1.1 Formal descriptions over natural-language descriptions

The way in which data formats are described and represented depends on the intended audience. Both *natural-language descriptions* for human engineers and *formal descriptions* for machine-processing exist:

- **Natural-language descriptions** intended for human engineers are still dominant at the time of writing. Translating data format knowledge from natural-language descriptions to machine-executable implementations, made necessary by the sheer volume of format-compliant data as well as by the complexity of its representation, depends on qualified human labour.

- **Formal descriptions** intended for machine-processing are currently present in specific, limited domains only. Universal applicability for describing arbitrary data formats in general depends on the availability of a formalised model that guarantees this property.

Accessing represented information in format-compliant data strongly depends on format-compliant implementations. Existing implementations are threatened by rapid technological change, which necessitates constant adaptation or replacement for them to remain usable. Yet, the problem with natural-language descriptions is that the application of data format knowledge to a problem depends on qualified human labour, which is limited by its cost and availability. Lacking format-compliant implementations, the illegibility of represented information effectively results in its loss.

Employing formal descriptions of data formats in a machine-processible manner is a desirable alternative for making data format knowledge machine-accessible and applicable without inherently depending on human labour. Yet, a suitable, formalised model still remains essential for the universal applicability of formal descriptions.

1.1.2 Declarative approaches over procedural approaches

The problem of formally describing data format knowledge can be approached in either a *procedural* or *declarative* manner:

- **In a procedural manner:** Using languages similar to existing programming languages like Java or C/C++, procedural approaches define fixed, specific algorithms for processing format-compliant data, such as for parsing and decoding, or for encoding and serialising. In contrast to declarative approaches, the underlying rules and constraints of format-compliant data are given only indirectly, albeit being present in the defined algorithms.

- **In a declarative manner:** Approaches using a declarative manner define the underlying rules and constraints that govern format-compliant data. In contrast to procedural approaches, no specific algorithm for processing format-compliant data is given.

For a specific purpose, procedural approaches make it straightforward to write algorithms to process format-compliant data. Yet, the problem with procedural approaches is that different purposes lead to different algorithmic implementations, even when the same underlying rules and constraints of the very same data format still apply.

Using a declarative approach, these underlying rules and constraints can be exposed for a wide range of applications, retaining the freedom of using the same format-specific knowledge for different purposes. Although more complex, the declarative approach has the benefit of increased flexibility.

For example, it may be desirable to access only a certain subset of information contained in large volumes of format-compliant data, thereby using resources more efficiently and speeding up the parsing process. Likewise, when part of format-compliant data is known to be corrupted, and thus not to be trusted to contain valid information, accessing the still-valid portion and trying to fix the corruption strongly depends on the availability of data format knowledge that can be adapted and applied in a flexible manner.

1.1.3 Research domains

The problem of describing data formats is of concern to data-centric domains of Computer Science, especially *Digital Preservation*, *Multimedia* and *Telecommunication*:

- In **Digital Preservation**, the problem of preserving long-term access to digital information for future generations threatens digitally represented cultural heritage [RH05]. A major use case for related work in this domain is the *migration of data* between data formats to prevent technological obsolescence and the subsequent loss of information.
- In **Multimedia**, two use cases related to data formats are the *normative definition of data formats* as well as the *high-level adaptation of digital objects* for *Universal Media Access* [VCE03]. Regarding the former use case, normative definition of data formats is required for the specification of new data formats, allowing the interoperability of systems working with multimedia data. Concerning the latter use case, Universal Media Access depends on the timely delivery of multimedia resources over heterogenous networks to end-user devices with varying decoding and playback capabilities. High-level, on-the-fly adaptation of digital objects to the capabilities and limitations of network and end-user devices strongly depends on data format knowledge to achieve meaningful adaptations.
- In **Telecommunication**, a primary use case is the *normative definition of protocol data units (PDUs).* Similar to Multimedia, there is the problem of describing communication protocols for interoperability of parties, which also covers the data format of PDUs to be transmitted.

Related work exists for formally describing data formats in a declarative and procedural manner, yet their underlying models are often based on domain-specific assumptions that do not hold in general. Examined related work does not provide a formalised model that is geared towards universal applicability, although universal applicability is sometimes claimed.

1.1.4 Data format instances and data formats

When considering the problem of formally describing data format knowledge, it is helpful to distinguish between a *data format instance* and a *data format*:

- **Data format instance:** A data format instance has a bit sequence, where information is represented conforming to its data format. Describing a data format instance in a machine-processible manner provides a means to authoritatively express how specific information is represented, correcting misunderstandings present in applications and in the understanding of human engineers. For example, a Portable Network Graphics (PNG) raster image is a data format instance for the PNG file format, where a description of this instance can show the exact bits responsible for stating the width of the given PNG image file.

- **Data format:** A data format has a potentially infinite number of its data format instances, which conform to a common set of underlying rules and constraints. Again, describing a data format in a machine-processible manner provides means to authoritatively express how a specific type of information is represented. For example, the PNG file format is a data format which defines rules on where the image header is located and from which fields it is composed, thus defining the placement of fields containing the width of arbitrary PNG images.

Data format instances and data formats are closely linked, since underlying rules and constraints of a data format manifest themselves in its data format instances. In order to address and consider both levels, it is desirable to have suitable, matching models for both describing data format instances as well as data formats.

1.2 Research Problems

This thesis addresses both the *formal description of arbitrary data formats in a declarative manner*, based on the *formal description of arbitrary data format instances*. Describing a data format as a class through its data format instances raises the following two research problems for this thesis:

- **Describing data format instances**: How to describe the composition of format-compliant data, considering the syntax and semantics of its bit sequence or segments thereof?
- **Describing data formats**: How to describe a data format with a potentially infinite set of data format instances through its underlying rules and constraints?

Since a data format serves the purpose of *representing information* for its storage and transmission over time, this thesis assumes that *the representation of information is lossless* - represented information must actually be present in its representation. This thesis therefore assumes as well that for every data format instance, there exists a *bijective mapping* between a format-compliant bit sequence and the information it represents.

1.3 Contributions and Outline

The four contributions of this thesis are a *state-of-the-art survey on data formats and their description*, an *analysis on data format description*, the *Bitstream Segment Graph (BSG) model for describing data format instances*, and the *BSG Reasoning approach for describing data format classes*:

- **Survey on current state-of-the-art in data formats and their description**: The survey in Chapter 2 covers definitions and provides a basic systematisation of related work in terms of their descriptive capabilities, focusing on the data-centric research domains of Digital Preservation, Multimedia and Telecommunication, and discussing shared properties, differences and shortcomings of related approaches.

- **Analysis of data format description**: The analysis provided in Chapter 3 presents an abstract model for describing data formats and data format instances in order to address the inherent properties and limitations of data format description. While the reversibility of a bijective mapping does not restrict its computational complexity, the analysis states that describing arbitrary data formats comes at the cost of losing guaranteed termination of parsing and decoding processes in case of erroneous data format rules. The analysis presents the concept of elementary descriptive capabilities, which align with the descriptive capabilities previously used for comparing related approaches and identifying shortcomings.
- **Bitstream Segment Graph (BSG) model**: Drawing on the set of elementary descriptive capabilities established in the analysis, the graph-based model presented in Chapter 4 is used to describe data format instances through a bijective mapping for a format-compliant bit sequence. The chapter includes methods for the construction of a BSG instance and its evaluation, e.g. for the extraction of contained information. The chapter also presents the Apeiron BSG Editor tool for the manual annotation of data and the definition of a RDF-based representation for BSG instances. This contribution has been published at the *International Conference on Software and Data Technologies (ICSOFT)* 2008 [HMT+08], at the *International Multi-Conference on Computing in the Global Information Technology (ICCGI)* 2008 [HSB+08] and extended in an article in the *International Journal on Advances in IT Security (IJAS)* [HFS+09], with the latter two focusing on applications in the context of IT Security.
- **BSG Reasoning approach**: The approach presented in Chapter 5 builds on the BSG model and describes a data format with a possibly infinite set of data format instances using rules for the computation of a least fixed point similar to Datalog [CGT89], thereby inferring a BSG instance from a format-compliant bit sequence. The chapter also presents a syntax for data format rules and a BSG-based reasoning engine. The contribution has been evaluated on a subset of the Portable Network Graphics (PNG) image format which exercises all elementary descriptive capabilities identified in the previous analysis. This contribution has been published at the *International Conference on Digital Preservation (iPRES)* 2008 [HBSM08] and extended in the IJAS article [HFS+09].

1.4 Acknowledgements

As Malcom Gladwell so aptly put it, behind every story of success, there is also a story of a fertile ground provided, of chances offered, of opportunities seized and of hard work being done even in the face of setbacks and constant frustrations.

Although writing a PhD thesis is a personal endeavour, it cannot be considered and understood as an isolated task. Looking back, I had a lot of helping hands when I needed them - in this spirit, I want to thank:

- my PhD advisor Prof. Dr. Max Mühlhäuser for taking the risk, accepting me as his PhD student, and giving me the freedom to pursue my lines of research in his TK / RBG group, as well as my PhD co-advisor Prof. Dr. Andreas Rauber for his rich, constructive feedback and a warm welcome to Vienna,

- my parents Astrid and Joachim and my sister Svea for keeping me grounded, for encouraging me all along the way, for supporting me in countless ways, and for listening to quite similar stories over and over again - finally, it is done -,
- all of my friends for being there, and for both distracting me and pushing me forward as it was necessary (especially, but neither limited to nor in a special order, Sandra and Hans, Tobias and Nina, Andreas and Verena, Marek and Steffi, Ariane, Maren, Kerstin, Annette and Claudia),
- my TK research team head Dr. Guido Rößling for a sound education in scientific procedures, and for his endless patience to both read and comment on countless drafts,
- my colleagues at the Telecooperation (TK) research group (especially Gina Häußge, Kai Höver, Henning Bär, and Gundolf von Bachhaus from the DLH group, as well as Sebastian Ries and others),
- my previous e-learning center (elc) team head Dr. Susanne Offenbartl for her support and her refreshingly pragmatic viewpoints on the better side of morale-building,
- my colleagues at the e-learning center (especially Irina Reuter, Anika Hartmann, Evelyn Ehrich, Klaus Steitz, Anne Bieberstein, Iris Wegmann, Thorsten Maas, Leif Pullich, Jens Japes and Julia Sonnberger) and at the Hochschulrechenzentrum (HRZ) (especially Dr. Jürgen Ohrnberger),
- my colleagues at the Rechnerbetriebsgruppe (RBG) group (namely Sven Schenkel, Claudia Kleber, Qun Herz, Manfred Schnitt, Gerhard Fladerer and Uwe Langendorff),
- my colleagues at the research training group GKEL (especially Doreen Böhnstedt)
- Prof. Dr.-Ing. Ralf Steinmetz and his KOM research group,
- all of my student helpers with whom I worked in close collaboration (especially Andreas Doms, Tim Klein, Hristo Lulev, Amir Neziri, Martin Pinto-Bazurco Mendieta, Viet Hung Tu and Thomas Wilker), as well as
- all of my students whose bachelor, diploma and master theses I supervised, and who contributed to my PhD research and my projects (namely in alphabetic order Jan Adler, Benedikt Antoni, Paul Baumann, Arsene Botchak, Alex Bulach, Julian Dean, Jens Hatlak, Clayton Hoss, Tim Klein (again), Benno Kröger, Friedrich-Daniel Möller, Bernd Papachrissanthou, Daniel Schumann, Marcus Ständer, Björn Stickler, Daniel Thies, Slaven Travar and Marc Weyland).

I probably still forgot half, and for that I am sorry - thanks to you all.

Chapter 2

State of the Art

2.1 Introduction

The previous Chapter 1 presented the research problem of formally expressing both the composition of format-compliant data for data format instances as well as the underlying rules for a data format in a machine-processible, declarative manner. In order to assess the current state of the art in this regard, this chapter now contributes a survey on the current state of the art regarding existing models for expressing such data format knowledge. The survey focuses on the following aspects:

- **Definitions and models for data format knowledge:** Related work directly concerned with data format knowledge regarding the composition of data gives definitions and models for its expression, either implicitly or explicitly. Existing definitions and models are based on their specific concepts and constraints, leading to inherent properties and limitations on their expressiveness.

- **References to existing related work:** Related work only indirectly concerned with the composition of data, but still in need to express such data format knowledge, provides references to existing definitions and models as well as insights to their adoption.

When related work provides a model for describing data formats or data format instances not on the level of meta-information, but focusing on data format rules and the structure of data format instances, the survey provides a *classification*:

- **Classification:** It is considered whether such an approach is *declarative or procedural*, whether it is *machine-processible* and whether it has a *formalised model for universal applicability*:

  - **Declarative or procedural approach:** In order to classify existing approaches for this thesis, it is considered whether they describe data format knowledge in a declarative or procedural manner.
  - **Machine-processible approach:** Approaches exist in related work which introduce descriptive means intended for human engineers, which do not focus on being machine-processible. Machine-processible approaches themselves require a minimum degree of formalisation which may be given only indirectly through textual definitions or executable implementations.

  - **Formalised model for universal applicability:** Approaches exist which introduce a number of concepts to provide specific descriptive capabilities, yet which do not address the completeness or orthogonality of their concepts for describing arbitrary data formats. The existence of such a formalised model supports potential claims for universal applicability.

Last but not least, if a suitable model is presented for an approach, the survey examines its *descriptive capabilities*:

- **Descriptive capabilities:** In order to compare different models, their expressivity is considered regarding the handling of *primitive data*, *structured data*, *transcoded data* or *fragmented data*:

  - **Primitive data** is a single piece of information, such as a floating-point number, a character string or a three-bit unsigned integer stored in least-significant bit first order. It is represented in an encoded form, which has to be decoded in order to access it.
  - **Structured data** is a continuous sequence of bit sequences, each with a separate, distinct "meaning" in its context. It has to be segmented to access the separate constituents.
  - **Transcoded data** is a bit sequence which is the result of a transformation of an original bit sequence, such as compression, encryption or some similar block transformation. It has to be transformed in order to access the original bit sequence.
  - **Fragmented data** is a bit sequence which is only a fragment of a larger, original bit sequence. In order to access the original bit sequence in its entirety, fragmented data has to be concatenated in the right order.

  Handling these kinds of data properly requires the matching *descriptive capabilities* of *decoding* primitive data, of *segmenting* structured data, *transforming* transcoded data as well as *concatenating* fragmented data are considered. Based on these descriptive capabilities, the survey compares related work, allowing statements to be made regarding their suitability for describing arbitrary data formats in general.

Since processing data is a central, recurring aspect of Computer Science, related work on data formats can be sought and found in a number of research domains. This chapter focuses on contributions from a data-centric subset of research domains in separate sections, namely *Digital Preservation* in Section 2.2, *Multimedia* in Section 2.3 and *Telecommunication* in Section 2.4. The chapter continues with a discussion in Section 2.5 and closes with a summary.

2.2 Digital Preservation

Overview

Digital Preservation is concerned with the *long-term preservation of digital information.* Data formats play a crucial role, since digital information is stored as data in a specific data format, while format-specific hardware and software provides access to the contained information. Over time, both hardware and software tend to become obsolete due to technological advances. As obsolete hardware may fail in the future when there is no replacement available, and as obsolete software may not be available anymore or fail to interoperate with newer hardware and software, *technological obsolescence* is a constant threat to Digital Preservation efforts. The current rapid pace of technological change amplifies this threat.

In literature, three preservation strategies for protecting digital information against information loss through technological obsolescence on the logical level are the *migration of data*, the *emulation of hardware / software*, and *digital archaeology*:

- **Migration of data**: Information stored as data in a specific data format is migrated to a suitable target data format. Due to a typical mismatch of different data formats, some information often cannot be retained during a migration and is usually lost. For a migration of data, it usually must be decided which information to retain, for which the migration process must be monitored [Arm00].

- **Emulation of hardware / software**: Hardware and/or software representing the original technological environment or parts thereof are replaced by an emulation. Through using the emulation, original hardware and/or software components remain capable of providing access to contained information [Rot99, Arm00]. A variation thereof is the virtualisation of software, where software for a specific technological environment is replaced with software that targets a virtual machine as an intermediate platform, which is available for the original technological environment. In case of technological obsolescence of the original environment, it is only necessary to port the virtual machine implementation to a new platform rather than porting each and every specific software. The Universal Virtual Computer serves as an example of such an approach [Lor01].

- **Digital archaeology**: Software, data and related documentation are analysed in order to reverse-engineer both syntax and semantics of data. Its description then serves as a basis for implementing new hardware or writing new software, which again provides access to contained information [RG99, Arm00].

The former three strategies depend on preparatory actions taken prior to the event of technological obsolescence. Without suitable preparation in advance, digital archaeology is the only remaining option, although usually costly [Wet98]. In contrast to the emulation of hardware / software, both migration of data and digital archaeology are concerned with data format knowledge on the composition of data.

Outline

The following sections explore the following related work in Digital Preservation with respect to the given preservation strategies:

- Maintaining format-related meta-information is addressed by *data format registries* such as the *US Library of Congress (LoC) Digital Preservation project*, the *Global Digital Format Registry (GDFR)*, *PRONOM*, and the recent merger of the latter two, the *Unified Digital Formats Registry (UDFR)*.

- Addressing issues related to the representation of information according to defined data formats, the *Open Archival Information System (OAIS)* Reference Model serves as a de-facto standard model regarding the long-term archival and preservation of data.

- For the migration of data, the selection and execution of suitable paths for conversion is a non-trivial problem. The *Typed Object Model (TOM)* is an approach for mediating data between different data formats in a distributed system, automating the process of migrating data between different formats.

- For measuring the quality of data migration in terms of retained information, the *eXtensible Characterisation Language (XCL)* project contributes both the *eXtensible Characterisation Extraction Language (XCEL)* for extracting information from data as a property, and the *eXtensible Characterisation Definition Language (XCDL)* for the description of such properties for later comparison.

- Related to aspects of Digital Preservation, but not explicitly framing itself into this domain, the *Data Format Description Language (DFDL)* is an approach for describing the composition of data according to a data format which focuses on leveraging existing XML technologies.

2.2.1 Data Format Registries

The preservation strategies previously presented in Chapter 2.2 depend on maintaining detailed meta-information on data formats and format-compliant applications in the long term. Therefore, there is a need for *data format registries* as custodians of meta-information on data formats.

Overview

Prominent data format registries are the *US Library of Congress Digital Preservation Project (LoCDP)* [AF05], the *Global Data Format Registry (GDFR)* [AS03], and the *PRONOM* [Bro05] data format registry. At the time of writing, both GDFR and PRONOM are in the process of merging to *Universal Data Format Registry (UDFR)* [UDF09].

Rich models for categorising data formats and for managing related meta-information in high detail are common to LoCDP, GDFR [Abr07b, Abr07a, AG08] and PRONOM [Bro05]. The model of GDFR extends to complex relationships between different data formats, such as extensions or versions, on a highly formal level. In varying forms, these registries provide means for referencing a specific data format, such as the *PRONOM Unique Identifier (PUID)* Scheme for data formats in the PRONOM registry.

Discussion

Data format registries allow their users to identify and associate data formats with metadata such as references to their specification, format-compliant applications or relations between formats required for the *migration of data* or *digital archaeology* in case of technological obsolescence. Examined related work provides definitions and models concerning format-related meta-information as well as references to other approaches.

- **Definitions and models:** Besides their rich models on format-related meta-data, all registries at least consider natural-language descriptions such as textual specifications suitable for human consumption as a baseline. In the case of PRONOM, format-related knowledge suitable for machine-processing is provided on the level of file signatures, allowing for the identification of a files' data format through automated tools such as *Digital Record Object Identification (DROID).*
- **References**: In terms of approaches for describing the composition of data, GDFR references to a number of format description languages. It explicitly refers to the *eXtensible Characterisation Extraction Language (XCEL)*, the *Bitstream Syntax Description Language (BSDL)* and the *Data Format Description Language (DFDL)* among other approaches, including formal grammar notations and XML schema languages.

2.2.2 Open Archival Information System Reference Model

Operating an archive for long-term archival and preservation of data poses a number of problems, which includes the management of how information is represented. Within short-term transactions, producers and consumers can negotiate the representation of information to be exchanged. Yet, for long archival and preservation, producers cannot foresee future representations of information, and thus need to delegate the negotation with future consumers to suitable archival systems, which manage a potentially required migration of data.

Overview

The Open Archival Information System (OAIS) provides a reference model for long-term archival and preservation systems, and serves as the de-facto standard regarding long-term archival and preservation of digital information [CCS02]. Among other aspects, OAIS addresses the issue of managing the representation of information.

The OAIS reference model defines a number of processes surrounding information stored in an OAIS archive between a *Producer* and its *Consumer*s, as well as the archive *Management.* Besides the processes of *Preservation Planning* and *Administration* that coordinate and manage archival operations between involved parties, actual processes related to archival and preservation are *Ingest*, *Archival Storage*, *Data Management* and *Access*, shown in Figure 2.1:

- **Ingest**: The Ingest process receives a *Submission Information Package* (SIP) or a later update thereof from a Producer, assuring its quality and generating an

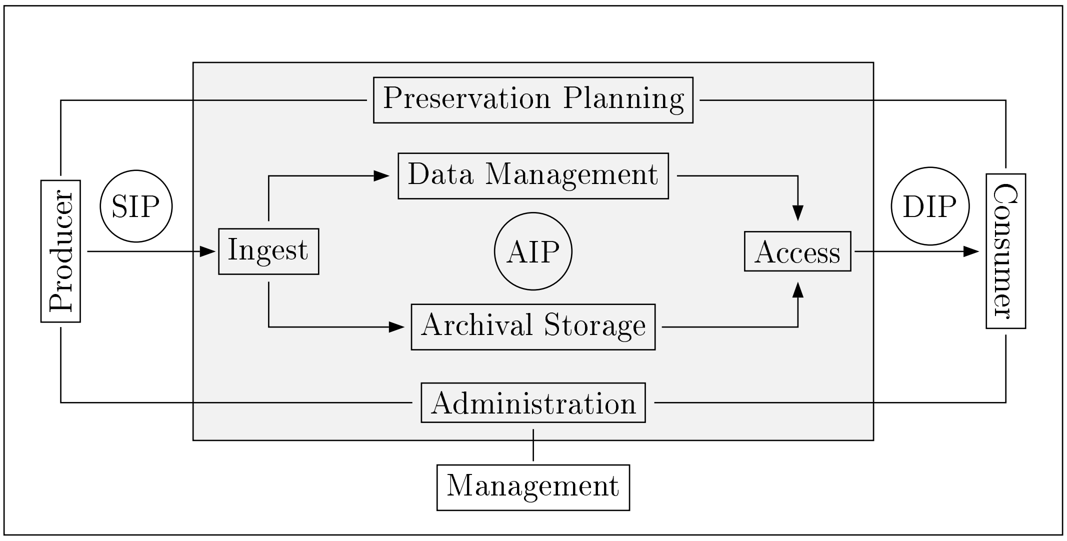


Figure 2.1: Overview of OAIS Functional Entities, based on [CCS02].

Archival Information Package (AIP) conforming to archive policies, such as using only publically disclosed data formats. It then generates descriptive infos and coordinates potential updates with Data Management.

- **Archival Storage**: The Archival Storage process is responsible for receiving AIPs to be put into storage, management of the storage, error checking and replacement of storage media, their backup for disaster recovery, and finally for providing an AIP upon request.
- **Data Management**: The Data Management process serves for administrating, updating and querying of archival databases as well as for general reporting for the archive.
- **Access**: Upon request by a Consumer, the Access process generates a suitable *Dissemination Information Package* (DIP) for the requested AIP and delivers it to the Consumer.

As both Archival Storage and Data Management are not central to the role of data formats in OAIS, these processes are mentioned for completeness and are not further explored.

As can be seen from the Ingest and Access process descriptions, the concepts of SIP, AIP and DIP are of importance to the OAIS reference model. These are specializations of the *Information Package*, being distinguished in its role in the respective process. As shown in Figure 2.2, an Information Package contains the *Content Information* as the actual content to be preserved, as well as related *Preservation Description Information.* The Content Information consists of a *Data Object* and its *Representation Information*, which can be used to obtain an *Information Object*:

- **Data Object**: A Data Object specialises into either a *Physical Object* as a physical representation, or a *Digital Object* as a digital representation through a set of bit sequences.

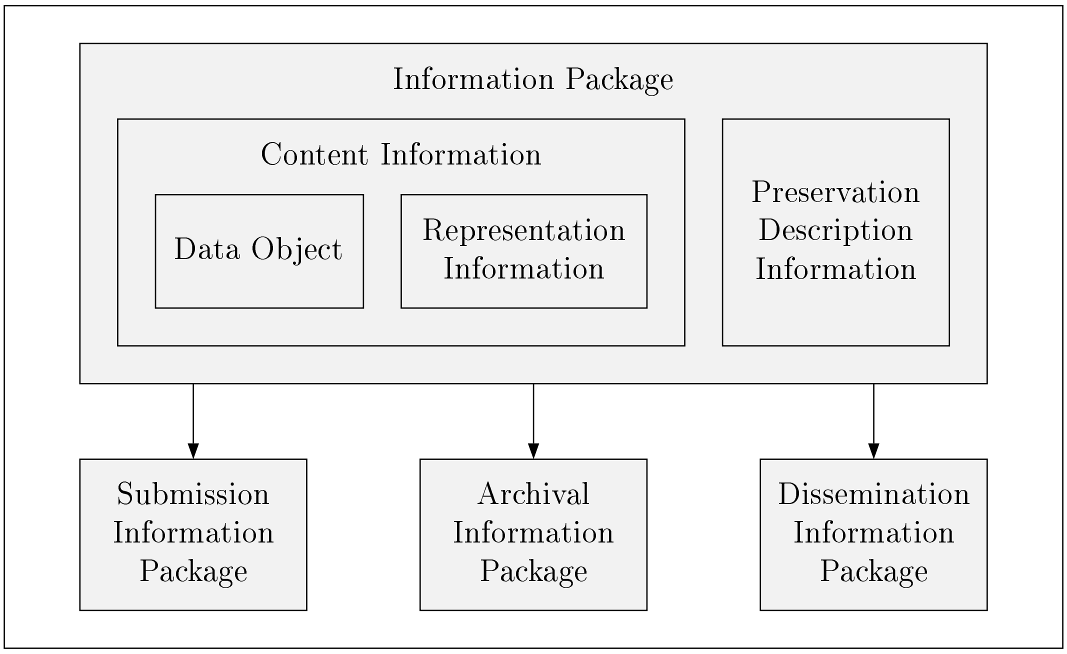

Information Package
Content Information
Data Object
Representation Information
Preservation Description Information
Submission Information Package
Archival Information Package
Dissemination Information Package

Figure 2.2: Overview of OAIS concept relations, based on [CCS02].

- **Representation Information**: Representation Information maps the Data Object to an Information Object as a “more meaningful concept”. Representation Information can be specialised into *Structure Information* which defines the mapping of bit sequences into data types, and into *Semantic Information* which defines the meaning of data. Representation Information itself may again be represented as a Data Object, and thus depends on other Representation Information, forming a *Representation Network* required for fully describing the meaning of an original Data Object.
- **Information Object**: An Information Object is obtained by interpreting a Data Object according to Representation Information.

Representation Information can be considered as a form of data format knowledge, which may be present in various forms such as textual descriptions in natural language, formal grammars, or some derivative work thereof, such as software implementations or even their source code.

Discussion

The OAIS reference model is intended for a long-term preservation archive, which also has to support the *migration of data* between different formats. The reference model thus depends on the availability and applicability of data format knowledge. It therefore refers to such knowledge in various forms as Representation Information, which is contained in Content Information in submission, archival and dissemination IPs.

- **Definitions and models:** The OAIS reference model does not provide an explicit, formal definition on data formats, but implicitly considers a data format as a definition of how information is represented. Furthermore, the OAIS reference model itself does not mandate a specific form for data format knowledge as Representation Information. It allows for a variety of ways to represent data format knowledge. The model explicitly mentions the option of "formal description languages containing well-defined constructs with which to describe data structures", referring to formalised approaches in general. Although the OAIS effectively is a de-facto standard in its domain, it does not provide more specific references or make a statement regarding the expressivity of existing approaches. For actually migrating data, an OAIS archive relies on software as a form of Representation Information to perform the migration.

2.2.3 Typed Object Model

Processing format-compliant data requires suitable, format-compliant applications. Although an application may be conceptually capable of processing a certain type of data such as video/audio recordings or text documents, it strongly depends on the specific digital representation to be parsed, decoded and processed. The diversity of different data formats thus can force users to mediate data between multiple data formats as necessary, therefore making it desirable to automate this process.

Overview

The *Typed Object Model (TOM)* serves for automating the process of migrating data between different formats in a distributed manner and has been published in the PhD thesis of John Ockerbloom [Ock98]. TOM defines both a *distributed system* and a *data model*:

- **Distributed system**: For mediating data between different data formats in a decentralised, scalable manner, TOM describes a distributed system consisting of agents that handle the processing and conversion of data in heterogenous data formats, operating on a distributed type graph. TOM defines the *Typed Object Protocol (TOP)* for communicating in such a distributed mediating system. The actual mediation is executed by *type brokers* as specialised agents, which offer their services and perform the actual processing on the behalf of clients. Other agents such as clients can query for type information, get attributes and call methods on objects or request conversions from a type broker.

- **Data model**: In the TOM data model, information is represented as an *object*, which is immutable and has both a value and a type. A *value* is not restricted to a digital representation such as finite byte sequences alone, but may also include abstract forms of representation.

  A *type* defines how the object and its value are to be interpreted. Here, an important type is the `ByteSeqType`. Objects of this type have finite byte sequences as values and thus can be stored and transmitted in a digital form. A type may define one or more attributes as well as one or more methods for its objects. An *attribute* extracts information from an object through a function without dependence on context information or additional parameters. A *method* derives information from an object through a function as well, yet may depend on context information or use additional parameters. TOM offers subtyping, so a type may have more specialised *subtypes*. It also enables *substitutability*, where given a type T, a subtype S of T and two objects t and s of types T and S, s can substitute for t, which allows objects to be considered at different levels of abstraction. The aggregated typing information defines a *type graph* on which TOM operates.

For obtaining different representations of the same information in TOM, an *encoding* describes a relationship between a pair of objects, the original *encoded object* and the resulting *encoding object* in a different representation. In order to cope with multiple, semantically equivalent encoding objects for a given encoded object, encodings in TOM are considered as one-to-many relations. Its inverse is a *decoding*, which is a many-to-one function. TOM considers encoding as the "refinement" of abstract objects. Likewise, it considers decoding as the "abstraction" of concrete objects.

Based on these concepts, TOM defines a *format* as a sequence of encodings to be applied on objects of a given type, which yields objects of the type `ByteSeqType`. A format allows to define a *shipped object*, which is an object including its format and thus allows a receiver to decode the object to the type indicated by the format.

For mediating data between different data formats in TOM, a *conversion* is a migration of data, which takes a shipped object as input and produces a shipped object as output. Often, a conversion between different formats cannot preserve all present information but only a subset. Therefore, a conversion tries to approximate the input shipped object. To manage the loss of information of such a conversion, TOM defines the concept of *intersubstitutability*, which is given for a conversion c and a type T if every input and output of the conversion c cannot be distinguished with regards to the attributes and methods of type T. The degree of information preservation in a conversion increases with every level down in the type hierarchy of T due to the specialisation of subtypes. Both substitutability and intersubstitutability aid in the automated composition and conversion of data between different formats in TOM.

The TOM approach has been implemented through the *Format REgistry Demonstration (FRED)*, which also served as a prototype for the *Global Data Format Registry (GDFR)* [Ock06].

Discussion

TOM uses format-related knowledge to assist the *migration of data* between different data formats. Contrary to other approaches, its main characteristic is the distributed setup of format-related knowledge among type brokers which provide migration services to other agents.

- **Definitions and models:** In TOM, information is considered in type-specific representations. The data format of such a type-specific information is defined as a sequence of encodings that converts information from its type-specific representation to its format-compliant byte sequences of `ByteSeqType` type, possibly using intermediate types.

Rather than describing the actual coding and structure of format-compliant data itself, it addresses the migration of data by describing encoding / decoding relationships between types. Through its use of interfaces, TOM considers digital objects on varying levels of abstraction, and is therefore able to indirectly manage the loss of information during a migration of data. External to TOM and its model, the actual process of accessing and conversion is performed by software tools which follow the underlying rules and constraints of a data format that shapes format-compliant data.

Rather than describing the composition of data format instances or the underlying rules and constraints of a data format itself, TOMs model exposes the type network composed from software operated in a distributed system. The "composability" as a property of TOM refers to the overall type network, and not to the composition of data.

2.2.4 eXtensible Characterisation Language

Technological obsolescence of file formats threatens long-term accessibility of contained information. Although data can be migrated in advance in order to prevent technological obsolescence, it typically leads to information loss due to mismatching representational capabilities of different data formats. One approach to handle such information loss is to identify *significant information* and to monitor its successful retention after a migration for it to be deemed successful.

Overview

For estimating the success of data migration, the *eXtensible Characterisation Language (XCL)* project defines the *eXtensible Characterisation Extraction Language (XCEL)* as well as the *eXtensible Characterisation Definition Language (XCDL)* for extracting and comparing significant information represented in different data formats:

- **eXtensible Characterisation Extraction Language (XCEL):** XCEL intends to describe characteristics of format-compliant data through *significant information*, which is represented as a property with a name and a value [SHC08]. Towards that goal, XCEL describes the composition of data through a XML-based, schema-like definition, where declarative definitions of data types are mixed with procedural processing instructions.

  The basic building blocks of XCEL are *XCEL elements* which are used to build an *XCEL Tree* as a representation which matches with actual data. A *symbol* element defines both the encoding and semantics of a byte sequence, may define constraints for matching and is a leaf of such a tree. A symbol has information on the placement of its data and employs an absolute addressing scheme, using the number of bytes consumed so far. An *item* element defines a logical, structural or semantic group for one or more child elements, describing either a sequence of elements, their permutation or a choice of alternative elements. Last but not least, a *processing* element allows the execution of methods in the XCEL processor, which provide means for placing an XCEL element elsewhere in the tree (`pushXCEL`), for copying another XCEL element to the current position (`pullXCEL`) or for reconfiguring the parser during its operation (`configureParser`). Moreover, processing elements also allow the addition of a *filter* to a *filter chain* of an element, allowing for the translation of data into another representation. Elements and their contents can be referenced through *identifiers* and may originate from a separate file (*externalSource*) or from another XCEL element (*internalSource*).

These XCEL elements are used in the schema-like *XCEL document*, which consists of the four parts `preProcessing`, `formatDescription`, `postProcessing` and `templates`. It is used by an *XCEL processor*, which processes both the XCEL document and an input file, and produces an XCEL Tree as *Result Tree*. In a following step, an *extractor* extracts significant information from the Result Tree which characterises the format-compliant data contained in the input file, and stores it as an XCDL document.

Example 2.2.1: An example of XCEL is given in Table 2.1 for the PNG IDAT chunk data structure, which carries compressed, transformed, and in some cases even fragmented, data representing the actual image. The data structure starts with the `chunkDataLength` symbol, which is an 32-bit unsigned integer, followed by the `pngIDATIdentifier`, which carries the four-byte ASCII string "IDAT" to distinguish it from other chunk data structures. The next processing statement sets the length of the yet-to-come `normDataSymbol` identifier to the value of `chunkDataLength`, so the `normDataSymbol` has a defined length. Last but not least, the remaining `crc` symbol carries a four-byte CRC value.

- **eXtensible Characterisation Definition Language (XCDL):** XCDL describes data as a collection of significant information that have been extracted previously from a Result Tree [BHST08].

  During the migration of data, some original input file is migrated to a new data format, producing a migrated file. After significant information has been extracted from both the original and the migrated file as XCDL documents, the degree to which the migration has been successful is measured through a comparison of the retained significant information. This is performed by a *comparator*, which processes both XCDL documents and compares its significant information through some domain-specific metric for judging the success of a migration.

Besides XCEL and XCDL, the XCL project also tries to address aspects of semantic mismatch between different XCDL documents through the definition of an *ontology*. Moreover, it tries to extend the comparison of significant information beyond the semantics of data to whether the actual rendering of data to human observers still carries the same significant information through the use of an *information model*.

Discussion

The use case of XCEL and XCDL is the evaluation to which degree a *migration of data* has been successful, based on the retainance of significant information within migrated data.

```
<item identifier="pngIDAT" xsi:type="structuringItem"
  multiple="true">
  <symbol identifier="chunkDataLength" interpretation="uint32"
    length="4"/>
  <symbol identifier="pngIDATIdentifier" interpretation="ASCII"
    optional="false" value="IDAT"/>
  <processing type="pushXCEL" xcelRef="normDataSymbol">
    <processingMethod name="setLength">
      <param valueRef="chunkDataLength"/>
    </processingMethod>
  </processing>
  <symbol identifier="normDataSymbol" interpretation="uint8"
    name="normData"/>
  <symbol identifier="crc" length="4"/>
</item>
```

Table 2.1: Excerpt of a XCEL description for a PNG IDAT chunk data structure, carrying transformed and compressed image data, taking from [SHC08].

- **Definitions and models:** The XCL project makes no explicit formal definition to what a data format actually is. Through the definition of XCEL, a quite complex model is given for describing a data format as a tree-based hierarchical representation of information. XCEL has a number of interesting properties, such as the support of *filters* for handling the translation between different representations of information, for *partial descriptions* which cover only part of an input file, or for allowing dependencies such as the placement of elements in the original data to be evaluated dynamically at runtime, which is of interest for address-based references in data formats such as the ISO Base File Format. Regarding XCDL, it is interesting to note that [BHST08] states it not to be intended as "a language for multi-purpose representation of information".

- **Classification:** The XCEL approach is both *declarative* and *machine-processible*, but the examined publications provide *no formalised model for universal applicability.*

- **Descriptive capabilities:** XCEL clearly provides support for segmenting structured data through *items* and for decoding primitive data through *symbols*, although the length of primitive data is limited to multiples of entire octets rather than having bit granularity. Due to its concept of *filters* used in a *filter chain*, and through using the `internalSource` attribute, XCEL can transform transcoded data and enable further processing of the result. Using the special *normData* symbol name for processing fragmented PNG IDAT chunk [SHC08], and again using the *internalSource* attribute, XCEL provides at least partial support for the concatenation of fragmented data, yet active control of fragment ordering is not explicit.

2.2.5 Data Format Description Language

For processing data in XML-based representations, a number of standardised technologies exist, such as transforming documents using Extensible Stylesheet Language Transformations (XSLT), or filtering elements using XPath. By translating the representation of data from the binary domain to XML, these technologies can be leveraged for use on data from the binary domain as well.

Overview

The *Data Format Description Language* (DFDL) is an extension to the W3C XML Schema Description Language (XSDL) and intends to describe arbitrary data formats to enable the translation from format-compliant data to an XML representation and vice versa. At the time of writing, the current version of DFDL is 1.0 as defined in Draft 038, where several parts of the language specification are designated to be in flux and to be changed in upcoming versions of the draft [PHB+10].

In the DFDL approach, a *processor* processes data given in a format as described by a *schema*:

- **Processor**: A processor typically is either a *DFDL Parser* or a *DFDL Unparser*, where the former parses the format-compliant representation and serialises it to an XML representation. Vice versa, the latter parses the XML representation and "unparses" its format-compliant binary representation, performing the reverse direction.

- **Schema**: A DFDL schema describes the composition of data through XML schema extended with DFDL annotations. While XML Schema provides the means to describe both primitive and complex data types, DFDL annotations describe additional information, such as the length or the binary encoding of a data type. For handling dynamic dependencies, where parsed and decoded information is used for further parsing, DFDL employs a subset of XPath 2.0 as expression language, including functions for boolean, string and date operations. Since DFDL has the explicit goal of *round-trip support* for data parsed and unparsed, every schema is required to be unambigious during unparsing, that is, only one binary representation may exist.

  EXAMPLE 2.2.2: A DFDL example is shown in Table 2.2, where structured data is described as a sequence of four primitive data, namely an integer `w`, an integer `x`, a double-precision floating-point number `y` and a single-precision floating-point number `z`, all in big-endian byte order.

Discussion

DFDL assists the *processing of data* for binary data formats through standardised XML tools by translating information from its format-compliant representation to an XML representation, and vice versa.

- **Definitions and models:** DFDL does not explicitly provide a formal definition of what a data format is, but indirectly defines its underlying model as tree-based

```
<xs:complexType name="example1">
  <xs:sequence>
    <xs:element name="w" type="int">
      <xs:annotation>
        <xs:appinfo source="http://www.ogf.org/dfdl/">
          <dfdl:element representation="binary"
             byteOrder="bigEndian"
             lengthKind="implicit"/>
        </xs:appinfo>
      </xs:annotation>
    </xs:element>
  <xs:element name="x" type="int ">
      <xs:annotation>
        <xs:appinfo source="http://www.ogf.org/dfdl/">
          <dfdl:element representation="binary"
            byteOrder="bigEndian"
            lengthKind="implicit"/>
        </xs:appinfo>
      </xs:annotation>
    </xs:element>
  <xs:element name="y" type="double">
      <xs:annotation>
        <xs:appinfo source="http://www.ogf.org/dfdl/">
          <dfdl:element representation="binary"
            byteOrder="bigEndian"
            lengthKind="implicit"/>
        </xs:appinfo>
      </xs:annotation>
    </xs:element>
  <xs:element name="z" type="float" >
      <xs:annotation>
        <xs:appinfo source="http://www.ogf.org/dfdl/">
          <dfdl:element representation="binary"
            byteOrder="bigEndian"
            lengthKind="implicit"
            binaryFloatRep="ieee" />
        </xs:appinfo>
      </xs:annotation>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

Table 2.2: Excerpt of a sample data structure defined using DFDL, taken from [PHB+10].

through the extension of XML Schema. It assumes that a data format defines the composition of "hierarchically-nested data", at the same time explicitly claiming its applicability on the description of any data format.

The DFDL specification includes two noteworthy concepts. It distinguishes between approaches for making data format knowledge explict as either *prescriptive* or *descriptive*. The specification document categorises approaches such as ASN.1 into the former category, and itself into the latter. Moreover, DFDL distinguishes between data as either *content* or *framing*, depending on its purpose in the format, where the language allows to "hide" framing from later processing.

- **Classification:** DFDL is a *declarative* approach which is *machine-processible.* In examined publications, *no formalised model for universal applicability* is given.

- **Descriptive capabilities:** DFDL clearly supports the segmentation of structured data and the decoding of primitive data, also supporting data with lengths of bit granularity through `dfdl:lengthUnits`. Although XML Schema is a powerful basis for DFDL to extend, concerning data format description, there are limitations to DFDL despite its explicit claim of universal applicability. Most notably, DFDL itself acknowledges its lack of support for cases where "one element's value becomes the representation of another element", termed "layering" by DFDL, which has been confirmed as a limitation and explicitly deferred to a later revision.

  Yet, layering is required for handling transformed or fragmented data, as in these cases, the value of one or more bit sequences represents another bit sequence when processed accordingly. For example, to completely describe video and audio streams typically stored as interleaved fragments in multimedia containers such as the MPEG-4 File Format, fragments of a specific stream have to be concatenated in order to analyse the stream according to its own format-specific rules, e.g. for MPEG-4 Video or MPEG-4 Advanced Audio Coding (AAC).

  Part of the lack can be attributed to DFDL's explicit assumption of data to be "hierarchically-nested", which fits well with the tree-based structural model of XML, where a logical node may have multiple children, but has at most one parent. Yet, for the concatenation of fragmented data, a logical node is required to have multiple parents as well, pointing towards a graph-based structural model.

  Therefore, it neither supports the descriptive capability to transform transcoded data, nor supports the descriptive capability to concatenate fragmented data.

2.3 Multimedia

Overview

The domain of Multimedia is a wide field of research that is concerned with digital multi-channel media. In literature, two primary drivers of format-related research in Multimedia have been the *normative definition of data formats* and the *high-level adaptation of digital objects*:

- **Normative definition of data formats:** Interoperability of multimedia systems requires involved parties to exchange information, and thus to mutually agree on the semantics of exchanged data. To provide a normative definition of data formats in a multimedia standard, a means of describing a data format is required.
- **High-level adaptation of digital objects**: Timely delivery of multimedia resources to end users over a network is usually restricted by resource-constrained networks and heterogenous capabilities of end-user devices. The vision of "Universal Media Access" [VCE03] addresses the issue by dynamically adapting multimedia resources as digital objects on-the-fly to given constraints, which requires machine-processible data format descriptions.

Outline

In the following subsections, the following approaches from both lines of research are introduced:

- For the definition of various multimedia-related standards by the Moving Pictures Expert Group (MPEG) such as MPEG-1 and MPEG-2, the so-called *MPEG 1/2 methodology* was used.
- In the later definition of the MPEG-4 standard, limitations of the MPEG 1/2 methodology lead to the design of the Syntactic Description Language (SDL), which later became the *Formal Language for Audio-Visual Object Representation (Flavor).*
- For enabling the high-level adaptation of digital objects, the MPEG-21 Part 7 standard on "Digital Item Adaptation (DIA)" describes the *Bitstream Syntax Description Language (BSDL).*
- Later work recombined the standardised BSDL approach with the expressiveness of Flavor, resulting in the twin *BFlavor* and *gBFlavor* approaches.

2.3.1 MPEG 1/2 Methodology

Among others, the MPEG-1 and MPEG-2 standards also define data structures intended for exchanging multimedia-related information such as video and audio between format-compliant systems. In this regard, these standard documents address system developers who are interested in creating or adapting interoperable systems, and who are in need of a uniform and unambigious convention for the description of these data structures.

Overview

With the term coined in [Ele97], the MPEG 1/2 methodology describes data structures in a tabular fashion:

- The first table column contains a mixture of pseudo-code statements resembling the programming language C, including `struct` field definitions, flow-control statements (eg. `if`, `while`) and special helper functions (eg. `nextbits()` for lookahead parsing), which guide the layout of fields containing data. Field definitions consist only of a name and have no type in this column.
- The second column contains the size of a field definition given as a fixed number of bits.
- The third column contains a mnemonic code which describes the encoding of data for a field definition, and thus provides its type. Examples of such codes are `uimsbf` (unsigned integer, most significant bit first) and `bslbf` (bit string, leftmost bit first).

Each field definition is necessarily located on a separate row in the table. Executing this pseudo-code on actual data in a cognitive walk-through step-by-step produces a consecutive layout of `struct` fields filled with data.

Example 2.3.1: An example of the MPEG 1/2 methodology is shown in Table 2.3, describing a `picture_header` data structure from MPEG-2 Video. This data structure consists of a sequence of fields such as the `picture_start_code` field with a length of 32 bits, encoded as *bit string, left bit first (bslbf)*, or the `picture_coding_type` field with a length of 3 bit, encoded as an *unsigned integer, most signficant bit first (uimsbf)*. Depending on the value of `picture_coding_type`, further fields such as the `full_pel_forward_vector` are present as well. Depending on the value of `extra_bit_picture` fields, multiple *extra_information_picture* fields may be present as well. The data structure ends with the `next_start_code()` function, which ignores padding zero bytes until the next MPEG-2 start code present in the stream.

Discussion

The MPEG 1/2 methodology enables the *normative definition of data formats* when it comes to static data structures documented for human engineers that are accustomed to working with pseudo-code representations.

- **Definitions and models:** Through a tabular form of pseudo-code, the MPEG 1/2 methodology allows the description of the composition of data to human engineers, who need to “execute” the description on a bit sequence in a mental walk-through. Most notably, the definition of fields is static regarding the mnemonic code expressing its type as well as its size in bits within a bitstream. Step-wise “execution” of pseudo-code implicitly manages the current position within the bitstream, and limits the resolving and placement of fields to a start-to-end order in a continuous sequence.

picture_header() {	**No. of Bits**	**Mnemonic**
picture_start_code	32	bslbf
temporal_reference	10	uimsbf
picture_coding_type	3	uimsbf
vbv_delay	16	uimsbf
if (picture_coding_type == 2 ||		
picture_coding_type == 3) {		
full_pel_forward_vector	1	bslbf
forward_f_code	3	bslbf
}		
if (picture_coding_type == 3) {		
full_pel_backward_vector	1	bslbf
backward_f_code	3	bslbf
}		
while (nextbits() == '1') {		
extra_bit_picture	1	uimsbf
extra_information_picture	8	uimsbf
}		
extra_bit_picture	1	uimsbf
next_start_code()		
}		

Table 2.3: Definition of the picture_header data structure from MPEG-2 Part-2 Video / ITU-T H.262 [IT95], using the MPEG 1/2 methodology.

- **Classification:** The approach is *procedural* due to its pseudo-code, although it is *not machine-processible*, and *no formalised model for universal applicability* has been given.

- **Descriptive capabilities:** A mental walk-through of a data format description in the MPEG 1/2 methodology traverses a sequence of typed fields, which is equivalent to segmenting structured data into a sequence of primitive data of variable length measured in bits, whose encoding is defined through the mnemonic code. Therefore, the MPEG 1/2 methodology supports both the segmentation of structured data and the decoding of primitive data. Despite the definition of pseudo-code procedures in this approach, it does not support the transformation of transcoded data, as the procedure serves for placing and accessing typed fields, but does not enable its actual transformation or further processing of its result. Likewise, the concatenation of fragmented data is not supported due to a lack of means.

2.3.2 Formal Language for Audio-Video Object Representation

During the definition of the MPEG-4 standards, *variable-length codes (VLCs)* such as the *Exponential Golomb integer encoding* were included into some of its data structures. As VLCs are variable in size, these data structures are dynamic, for which the MPEG 1/2 methodology is not sufficient [Ele96]. Therefore, an improved approach for data format description became necessary for use in the MPEG-4 standards.

Overview

The *Syntactic Description Language (SDL)* was initially proposed as a new language for describing dynamic data structures for use in MPEG-4 [Ele95]. SDL was included in the *MPEG-4 Systems and Description Languages (MSDL)* [ACE+97] and was later renamed as *Formal Language for Audio-Visual Object Representations (Flavor)* [Ele97]. *XFlavor* is an extension to Flavor, which enables the translation of data between a format-specific, binary representation and an XML-based representation [EH02, HE08]. Examples for both Flavor and XFlavor usage are given below.

Two primary goals of Flavor are the description of dynamic data structures as well as the separation of parsing from decoding, explicitly limiting the focus of the language to parsing alone [Ele96]. While the former goal addresses data structures from MPEG-4 using VLCs, the latter shall enable data structures to be adaptable. An example that is given explicitly is being able to change the number of bits used for representing a value without having to change the actual decoding algorithm [Ele97].

As stated, the Flavor approach provides a *language* for describing data structures. Not unlike the MPEG 1/2 methodology in terms of using pseudo-code for description, the Flavor language mixes declarative definitions of field types (e.g. `unsigned integer(32)`) with procedural statements for flow-control (e.g. `if`, `while`). Beyond the capabilities of the MPEG 1/2 methodology, it adds support for variable field

```
aligned(8) class Box (unsigned int(32) boxtype,
  optional unsigned int(8)[16] extended_type) {
    unsigned int(32) size;
    unsigned int(32) type = boxtype;
    if (size==1) {
        unsigned int(64) largesize;
    } else if (size==0) {
        // box extends to end of file
    }
    if (boxtype=='uuid') {
        unsigned int(8)[16] usertype = extended_type;
    }
}

aligned(8) class FileTypeBox extends Box('ftyp') {
    unsigned int(32) major_brand;
    unsigned int(32) minor_version;
    unsigned int(32) compatible_brands[];
}
```

Table 2.4: Definition of a Box and a File Type Box using MSDL / Flavor from the ISO Base File Format [ISO05a].

sizes and introduces object-oriented concepts such as classes and their inheritance as known from other programming languages, so that data structures are effectively defined as classes in Flavor [DNVDDS+06]. Using the `flavorc` translator, Flavor source code can be compiled to source code implementing parsers and serialisers in either Java or C++, providing in-memory representations of binary, format-compliant data [DNVDDS+06]. XFlavor extends this approach towards XML-based representations, embedding data within the document which can be processed further using established XML standards [HE08].

Example 2.3.2: Two Flavor descriptions are shown in Table 2.4 and 2.5, which describe the FileTypeBox data structure from the ISO Base File Format as well as the System Header data structure from the MPEG-2 Program Stream (PS) format. When actual Stream Header data from a MPEG-2 Program Stream is to be processed, the first field in its class definition in Table 2.5 is the `start_code` field, which is of type `unsigned int(32)`, an unsigned integer of 32 bit length. When this Flavor class definition is translated into Java using `flavorc`, it results in a Java class, representing a System Header that has been read from binary data and can again be written, including an in-memory representation of the actual value of the `start_code` field. With XFlavor, the resulting Java class can also be used to write an XML representation of itself, which would then contain a `start_code` XML tag containing the actual field value formatted as a string. Table 2.6 shows an example of such an XML representation for the SystemHeader class definition from Table 2.5.

```
class SystemHeader {
  unsigned int(32) start_code;
  unsigned int(16) header_length;
  bit(1) marker = 0b1;
  unsigned int(22) rate_bound;
  bit(1) marker = 0b1;
  unsigned int(6) audio_bound;
  bit(1) fixed_flag;
  bit(1) csps_flag;
  bit(1) sys_aud_lock_flag;
  bit(1) sys_vid_lock_flag;
  bit(1) marker = 0b1;
  unsigned int(5) vid_bound;
  bit(1) pkt_rate_restr_flag;
  const bit(7) reserved = 0x7F;
  while (nextbits(1) == 0b1) {
    unsigned int(8) stream_id;
    bit(2) bit_pattern = 0b11;
    bit(1) buff_bound_scale;
    unsigned int(13) buff_size_bound;
  }
}
```

Table 2.5: Definition of a MPEG-2 Systems Program Stream (PS) using Flavor, taken from [HE08].

```
<system_header>
  <start_code>443</start_code>
  <header_length>12</header_length>
  <marker>1</marker>
  <rate_bound>7653</rate_bound>
  <marker>1</marker>
  <audio_bound>1</audio_bound>
  <fixed_flag>1</fixed_flag>
  <csps_flag>0</csps_flag>
  <sys_aud_lock_flag>1</sys_aud_lock_flag>
  <sys_vid_lock_flag>1</sys_vid_lock_flag>
  <marker>1</marker>
  <vid_bound>1</vid_bound>
  <pkt_rate_restr_flag>1</pkt_rate_restr_flag>
  <reserved>127</reserved>
  <stream_id>224</stream_id>
  <bit_pattern>3</bit_pattern>
  <buff_bound_scale>1</buff_bound_scale>
  <buff_size_bound>80</buff_size_bound>
  <stream_id>192</stream_id>
  <bit_pattern>3</bit_pattern>
  <buff_bound_scale>0</buff_bound_scale>
  <buff_size_bound>80</buff_size_bound>
</system_header>
```

Table 2.6: XML representation of a MPEG-2 Systems Program Stream (PS) obtained through XFlavor using the Flavor description in Table 2.5, taken from [HE08].

Discussion

By using the Flavor language, a number of dynamic data structures can be described through object-oriented code which can be translated into Java or C++ components suitable for parsing and serialising format-compliant data to and from in-memory representations. Using the XFlavor variant, format-compliant data can be mapped from binary to XML-based representations for further processing.

- **Definitions and models:** The Flavor approach provides a language for describing the composition of data through object-oriented source code that can process format-compliant data. Although initially claimed to be "declarative", Flavor describes the layout of data through procedural code including flow-control statements and look-ahead parsing operators. In contrast to a declarative approach, code in the Flavor language defines *how* data is accessed rather than describing *what* data is present where.

  Regarding its expressiveness, the Flavor language has been explicitly limited to parsing in its early stages through the explicit separation of parsing from decoding and its focus on parsing. The limitation is detailed and argued for by its authors via the introduction of the *problem of high-level context* [Ele96], where parsing a conditional data structure would depend upon a decoded primitive data value. Overcoming this limitation by extending and generalising the Flavor language is dismissed as "not useful", as a survey by the authors on several multimedia specifications such as H.263 or MPEG-2 Video did not exhibit a case of this problem.

  Another noteworthy aspect is that XFlavor has drawn some criticism for its verbosity, in part through its use of XML and in part through embedding binary data into the XML document itself, leading to questions regarding a suitable granularity of description [DNVDDS+06].

- **References:** Flavor makes the connection between parsing and serialising methods as some sort of "marshalling" between representations and thus connects it with the *External Data Representation (XDR)* standard, the CORBA *Interface Definition Language (IDL)* and even the *Abstract Syntax Notation One (ASN.1)*.

- **Classification:** Flavor provides a *procedural*, *machine-processible* approach for describing data formats, yet it presents *no formalised model for universal applicability* in literature.

- **Descriptive capabilities:** Similar to the MPEG 1/2 methodology, Flavor supports the segmentation of structured data as well as the decoding of primitive data with bit granularity. Since Flavors flow-control statements guide the processing of format-compliant data rather than enabling actual transformation, it does not support the transformation of transcoded data or the concatenation of fragmented data.

2.3.3 Bitstream Syntax Description Language

In Multimedia, the vision of Universal Media Access (UMA) requires on-demand adaptation of digital items to current constraints as posed by the network and

```
<?xml version="1.0"?>
<xsd:schema targetNamespace="JP2"
 xmlns:bsdl-1="urn:mpeg:mpeg21:2003:01-DIA-BSDL1-NS"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 elementFormDefault="qualified">

  <xsd:import
   namespace="urn:mpeg:mpeg21:2003:01-DIA-BSDL1-NS"
   schemaLocation="BSDL-1.xsd"/>

  <xsd:element name="SOP">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Marker" type="xsd:hexBinary"/>
        <xsd:element name="LMarker" type="xsd:unsignedShort"/>
        <xsd:element name="Nsop" type="xsd:unsignedShort"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="PacketData" type="bsdl-1:byteRange"/>
  ...
</xsd:schema>
```

Table 2.7: Definition of a sample BS Schema for a JPEG2000 bitstream using BSDL [PHH+03].

terminal device. On-demand adaptation of digital items is a non-trivial problem, as digital items stored today may need to be adapted to future requirements in a yet unforeseen way. Suitable means for enabling on-demand adaptation of digital objects are thus desirable.

Overview

The Bitstream Syntax Description Language (BSDL) has been specified by the Moving Pictures Expert Group (MPEG) as part of the MPEG-21 Digital Item Adaptation (DIA) framework defined in ISO/IEC 21000-7. For reuse in other contexts, it has also been defined separately in ISO 23001-5. The approach has been the subject of several books and publications [Dev03, BPVdWK06, PHH+03]. The goal of BSDL is to enable generic, interoperable adaptation engines for adapting digital items in different data formats. BSDL assumes data formats to support the notion of a *scalable bitstream*, where adaptations can be generated through simple filtering rather than format-specific or computationally complex transformations.

The focus of the BSDL approach is on processing a bitstream through its *bitstream syntax (BS)* by extending standards such as XML and XML Schema. Using BSDL, a *BS schema* can be defined, which can then be applied to a bitstream to obtain a format-specific XML-based *BS description* which refers to the bitstream.

```
<?xml version="1.0"?>
<Codestream xmlns="JP2" xmlns:jp2="JP2"
 xsi:schemaLocation="JP2 JP2.xsd">
  ...
  <Bitstream>
    <Packet>
      <SOP>
        <Marker>FF91</Marker>
        <LMarker>4</LMarker>
        <Nsop>0</Nsop>
      </SOP>
      <PacketData>155 242</PacketData>
    </Packet>
    ...
  </Bitstream>
</Codestream>
```

Table 2.8: Definition of a sample BS Description for a JPEG2000 bitstream using BSDL [PHH+03].

In contrast to format-specific BS descriptions, BSDL also allows the use of a generic XML representation for a *generic BS (gBS) description*, where generating the gBS description through parsing is application-specific and serialising it to a bitstream depends on a uniform *gBS Schema*, where structured data is represented through `gBSDUnit` nodes. Its simplification to a uniform schema reduces the complexity compared to arbitrary BS schemata, thereby enabling the use of gBS descriptions for content adaptations on resource-constrained devices.

A BS description can be used for adaptation in the DIA framework, where adaptation is considered as a three-stage process consisting of *parsing the original bitstream*, *transforming the BS description* depending on the desired adaptation, and *generating the adapted bitstream* from the transformed BS description and the original bitstream itself.

For executing the parsing and generation stages, the DIA framework defines the `BintoBSD` parser (for parsing binary data) as well as the `BSDtoBin` parser (for serialising binary data) as components, which operate on a given BS schema. Both components map between a binary, format-specific representation and an XML-based representation. The transformation stage for adaptation itself is not specifically mandated by DIA itself, but has been repeatedly addressed by the use of *eXtensible StyleSheet Language Transformation (XSLT)* as a standardised approach for transforming XML-based representations. The possibility of using alternative approaches such as the *Streaming Transformations for XML (STaX)* is suggested as well.

BSDL itself extends XML Schema on two *levels*, termed *BSDL-1* and *BSDL-2*. BSDL-1 provides required information for generating a bitstream from a BS description. BSDL-2 builds upon BSDL-1 by providing required information for parsing a bitstream to a BS description as well. In order to generate an adaptation from an original bitstream, the BSDL approach provides a linear addressing scheme

```
<?xml version="1.0"?>
<dia:DIA xmlns="urn:mpeg:mpeg21:2003:01-DIA-gBSD-NS"
 xmlns:dt="urn:mpeg:mpeg21:2003:01-DIA-BasicDatatypes-NS"
 xmlns:gbsd="urn:mpeg:mpeg21:2003:01-DIA-gBSD-NS"
 xmlns:dia="urn:mpeg:mpeg21:2003:01-DIA-NS"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="urn:mpeg:mpeg21:2003:01-DIA-gBSD-NS
   gBSSchema.xsd urn:mpeg:mpeg21:2003:01-DIA-BasicDatatypes-NS
   BasicTypes.xsd">

  <dia:Description xsi:type="gBSDType">
    ...
    <gBSDUnit syntacticalLabel=":J2K:MainHeader" start="0"
     length="135">
      <gBSDUnit syntacticalLabel=":J2K:SOC" start="0"
       length="2"/>
      <gBSDUnit syntacticalLabel=":J2K:SIZ" start="2"
       length="49">
        <Header>
          <DefaultValues addressMode="Consecutive"/>
        </Header>
        <Parameter name=":J2K:Marker" length="2">
          <Value xsi:type="xsd:hexBinary">FF51</Value>
        </Parameter>
        <Parameter name=":J2K:Lsiz" length="2">
          <Value xsi:type="xsd:unsignedShort">47</Value>
        </Parameter>
        <Parameter name=":J2K:Rsiz" length="2">
          <Value xsi:type="xsd:unsignedShort">0</Value>
        </Parameter>
        <Parameter name=":J2K:Xsiz" length="4" marker="R">
          <Value xsi:type="xsd:unsignedInt">768</Value>
        </Parameter>
        <Parameter name=":J2K:Ysiz" length="4" marker="R">
          <Value xsi:type="xsd:unsignedInt">512</Value>
        </Parameter>
        ...
      </gBSDUnit>
      ...
    </gBSDUnit>
    ...
  </dia:Description>
</dia:DIA>
```

Table 2.9: Definition of a sample gBS Description for a JPEG2000 bitstream using BSDL [PHH+03].

to allow BS descriptions to provide references to a bitstream or portions thereof. It is even possible for a BS description to represent a bitstream or parts thereof as well, by embedding binary data through Base64 encoding in XML. Yet, as encoding binary data in Base64 results in an 1/3 increase in size, the resulting verbosity usually penalises such a design choice.

The DIA framework does not mandate the structure or granularity of BS descriptions for a specific data format, and leaves these choices to the designer of a BS schema. Related to that, [BPVdWK06] states that the descriptive granularity of BS schemata and BS descriptions are scalable and only depend on the needs of a specific application. The authors argue that describing binary data on a bit-by-bit granularity is rarely necessary and typically too large for use and exchange, due to the inherent verbosity of the XML-based description. They therefore assume that a high-level description is usually sufficient for adaptation.

EXAMPLE 2.3.3: Excerpts of a BS schema, a BS description and a gBS description in the context of JPEG 2000 are given as examples in Tables 2.7, 2.8 and 2.9, respectively. The BS schema excerpt in Table 2.7 defines a so-called *start-of-packet (SOP) marker* as a sequence of fields called `Marker`, `LMarker` and `Nsop`, which are a two-byte hexadecimal value and two unsigned short integer values. The corresponding BS description excerpt in Table 2.8 shows a start-of-packet marker with actual primitive data values. Finally, the gBS description excerpt in Table 2.9 shows the "main header" of a JPEG 2000 image including its contained data structures as `gBSDUnit` nodes. This data structure is described as a sequence consisting of an opaque *start-of-codestream (SOC) marker* data structure, followed by an *image and tile size (SIZ) marker* data structure. Contained in the SIZ marker data structure, its `Xsiz` and `Ysiz` fields represent primitive data as four-byte unsigned integers, stating that the described main header belongs to a JPEG 2000 image which has a resolution of 768×512 pixels.

Discussion

The BSDL approach enables high-level content adaptation of scalable bitstreams by mapping data from binary to XML-based representations, performing simple filtering operations on the XML representation by using XML processing standards, and mapping it back to the binary domain again.

- **Definitions and models:** BSDL provides an approach based on XML Schema for describing composition of data given in data formats which follow the scalable bitstream assumption. Regarding general applicability, [Dev03] states that BSDL is not intended for parsing and decoding arbitrary bitstreams completely, citing examples such as entropy coding, wavelet coding or Discrete Cosine Transformation (DCT). The author argues that as most coding formats have been specified without the use of a formal language, they do not follow any constraints in this regard. The author furthermore argues that major parts of a bitstream are the result of an encoding process that is not within the scope of the BSDL approach for data format description.

  Also interesting to observe is that although BSDL provides definitions for a generic BS description (gBSD) in XML, the generation of gBSD is application-specific and left to format-specific applications outside the scope of BSDL [PHH+03]. Giving a justification, [VDDNDSVdW08] states that parsing a bitstream to a gBS description in a format-agnostic way is difficult due its dependency on the specific type of adaptation and its dependency on the format as such.

- **Classification:** BSDL provides a *declarative* approach that is *machine-processible*. In examined literature, *no formalised model for universal applicability* is presented.
- **Descriptive capabilities:** Not unlike the DFDL approach, BSDL supports the segmentation of structured data as well as the decoding of primitive data, where data may have a length measured in bits, due to the `addressUnit` attribute. Due to the focus of BSDL on scalable bitstream for high-level content-adaptation through simple filtering rather than performing computationally complex transformations, no support for the transformation of transcoded data is provided. Likewise, the concatenation of fragmented data is not addressed as well.

2.3.4 BFlavor and gBFlavor

Using the BSDL approach for describing the composition of data formats providing for scalable bitstreams requires the declaration of BSDL schemata. Compared to procedural approaches such as Flavor, writing BSDL schemata can be complex and is less expressive. As the existing BintoBSD reference implementation has been shown to be inefficient for processing large volumes of format-specific data, it is desirable to automatically generate and use more efficient format-specific parsers.

Overview

The BFlavor approach combines and extends the previously defined Flavor and BSDL approaches [DN]. From Flavor, it uses its procedural definition regarding the composition of data, provides for the automatic generation of source code for a parser implementation as well as a corresponding BS schema in the BSDL language. The variant gBFlavor is similar to BFlavor, but focuses on gBS descriptions rather than BS descriptions.

Since for both BFlavor and gBFlavor, the initial stage of generating a BS description from a given bitstream is delegated to automatically generated implementations, they extend BSDL only on the level BSDL-1 (for generating adapted bitstreams from BS descriptions), but not on BSDL-2 (for generating BS descriptions from bitstreams).

Discussion

As with the BSDL approach, BFlavor and gBFlavor enable high-level content adaptation of scalable bitstreams through using XML processing standards. Mapping data from its binary representation to XML is improved over BSDLs generic `BintoBSD` parser through the generation of format-specific parsing components.

- **Definitions and models:** Essentially, the (g)BFlavor approach describes the same approach for describing the composition of data that Flavor provides. They therefore inherit Flavor's implicit management of placement and positioning during parsing, based on the forward execution of the (g)BFlavor code. Moreover, both approaches inherit its assumption of data formats to support scalable bitstreams.

- **Classification:** As with Flavor itself, its (g)BFlavor extension is a *procedural*, *machine-processible* approach, for which *no formalised model for universal applicability* is presented in examined literature.
- **Descriptive capabilities:** The (g)BFlavor approach provides the same descriptive capabilities as Flavor, supporting the segmentation of structured data and the decoding of primitive data with bit granularity, but without support for the transformation of transcoded data or the concatenation of fragmented data.

2.4 Telecommunication

Overview

Similar to digital objects from Multimedia, the need for interoperability has lead to format-related efforts in Telecommunication on the *normative definition of protocol data units (PDUs)*:

- **Normative definition of protocol data units (PDUs):** In a protocol exchange, PDUs are exchanged as messages between involved parties. Since these parties need to agree to and understand the implied semantics of these messages, their composition has to be described for documentation.

Outline

In the following sections, the two following approaches are considered as related work for this thesis. Both address the normative description of PDUs used in telecommunication protocols:

- Standardised by the International Telecommunication Union, the *Abstract Syntax Notation One (ASN.1)* is a well-known approach for a machine-processible description of messages, which can be combined with the *Encoding Control Notation (ECN)* for encodings that are not provided by the predefined ASN.1 codecs.
- Conceived by the *European Telecommunications Standards Institute (ETSI)* as a simpler alternative to the highly complex specifications of ASN.1 and ECN, the *Concrete Syntax Notation 1 (CSN.1)* serves to describe the representation of messages on the bit level.

2.4.1 Abstract Syntax Notation One

An inherent and ever-present need in Telecommunications is to define the messages to be transmitted in a new protocol exchange. It is desirable to describe these messages on a high level and to delegate the corresponding definition of their representation as bits and bytes to standardised, reusable codecs which solve common problems such as variable-length fields in a uniform manner.

Overview

The *Abstract Syntax Notation One (ASN.1)* is a set of specifications concerned with specifying messages for protocol exchanges that have been standardised by the *International Telecommunication Union Telecommunication Standards Sector (ITU-T)*. Analogous to the distinction between Application and Presentation layer on the OSI network model, ASN.1 distinguishes between the *abstract syntax* and the *transport syntax* of a message:

- **Abstract syntax:** The abstract syntax of a message is concerned with its composition from typed fields that carry information with defined semantics, but does not mandate a specific encoding. Using the ASN.1 *language*, so-called *modules* contain message definitions as *assignments*, such as shown for X.509 certificates in Table 2.10. In this example, the `Certificate` type is defined as a sequence of three fields `tbsCertificate`, `signatureAlgorithm` and `signatureValue` of types `TBSCertificate`, `AlgorithmIdentifier` and `BIT STRING`, respectively. Likewise, the `Time` type is defined to allow a choice between carrying a field `utcTime` of type `UTCTime` or a field `generalTime` of type `GeneralizedTime`.

The ASN.1 language provides support for built-in primitive types (such as BitString, CharacterString or Integer) and complex types (such as SEQUENCE, SET or CHOICE) which can also be constrained and composed to form user-defined types [IT97]. Furthermore, the ASN.1 language provides support for concepts such as for enabling the *layering* of messages in a protocol stack, and for enabling the *extensibility* of messages to enable at least partial interoperability between different versions of messages [Lar99].

- **Transport syntax:** The transport syntax of a message is concerned with the way its fields are encoded to a binary representation of the message. For a given abstract syntax and message, its corresponding transport syntax is obtained either by reusing encoding rules of existing ASN.1 *codec*, or defining and using new encoding rules through the *Encoding Control Notation (ECN)* language.

  Existing ASN.1 codecs include the *Basic Encoding Rules (BER)* codec [IT02a] which is relatively simple, the *Packed Encoding Rules (PER)* codec [IT02b] which is compact but more complex in processing, or the *XML Encoding Rules (XER)* codec [IT01] which produces XML-based representations.

  The ECN language serves for defining alternative encodings that are variants of existing codecs or new encodings altogether [IT02c]. Analogous to the ASN.1 language, ECN provides built-in encoding classes for primitive ASN.1 types (such as #INTEGER for the ASN.1 type INTEGER) and complex ASN.1 types (such as #SEQUENCE or #CHOICE). Moreover, built-in encoding classes also include encoding procedures (such as #TRANSFORM) that allow transformation of values between different encoding classes like #CHAR and #INTEGER, and provide support for arithmetic operations. In a process termed *coloring*, the implicitly defined ECN types from an ASN.1 definition are recursively replaced until an explicitly generated encoding structure is produced of the abstract syntax of the message, with which the message can be encoded.

  EXAMPLE 2.4.1: An example of ECN use is the encoding object assignment shown in Table 2.12 which defines how the ASN.1 assignment in Table 2.11, a `PositiveIntegerBCD`, is to be encoded using the `positiveIntegerBCDEncoding` encoding. According to Table 2.12, the integer value is converted into a sequence of characters, which is then encoded using the `numeric-chars-to-bcdEncoding` encoding. This encoding is aligned to nibble (four-bit) boundaries, where each character of the character sequence is mapped to a four-bit sequence, which is finally appended with a terminating bit sequence "1111".

ASN.1 has been used in the definition of file formats as well as messages in network protocols. Examples are the file format for X.509 certificates [CSF+08] storing cryptographic information, as well as messages from the H.323 protocol suite used in

```
Certificate  ::=  SEQUENCE  {
        tbsCertificate       TBSCertificate,
        signatureAlgorithm   AlgorithmIdentifier,
        signatureValue       BIT STRING  }

TBSCertificate  ::=  SEQUENCE  {
        version         [0]  EXPLICIT Version DEFAULT v1,
        serialNumber         CertificateSerialNumber,
        signature            AlgorithmIdentifier,
        issuer               Name,
        validity             Validity,
        subject              Name,
        subjectPublicKeyInfo SubjectPublicKeyInfo,
        issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
                             -- If present, version MUST be v2 or v3
        subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,
                             -- If present, version MUST be v2 or v3
        extensions      [3]  EXPLICIT Extensions OPTIONAL
                             -- If present, version MUST be v3
        }

Version  ::=  INTEGER  {  v1(0), v2(1), v3(2)  }

CertificateSerialNumber  ::=  INTEGER

Validity ::= SEQUENCE {
        notBefore      Time,
        notAfter       Time }

Time ::= CHOICE {
        utcTime        UTCTime,
        generalTime    GeneralizedTime }
```

Table 2.10: Excerpt of a X.509 certificate definition given in ASN.1 from [CSF+08].

```
PositiveIntegerBCD ::= INTEGER(0..MAX)
```

Table 2.11: An ASN.1 assignment from [IT02c] for use in conjunction with the ECN encoding object assignment shown in Table 2.12.

```
positiveIntegerBCDEncoding #PositiveIntegerBCD ::= {
  USE #CHARS
  MAPPING TRANSFORMS{{
  INT-TO-CHARS
  -- We convert to characters (e.g., integer 42
  -- becomes character string "42") and encode the characters
  -- with the encoding object "numeric-chars-to-bcdEncoding"
  SIZE variable
  PLUS-SIGN FALSE}}
  WITH numeric-chars-to-bcdEncoding }

numeric-chars-to-bcdEncoding #CHARS ::= {
  ALIGNED TO NEXT nibble
    TRANSFORMS {{
      CHAR-TO-BITS
        -- We convert each character to a bitstring
        --(e.g., character "4" becomes '0100'B and "2" becomes '0010'B)
        AS mapped
        CHAR-LIST { "0","1","2","3",
          "4","5","6","7",
          "8","9"}
        BITS-LIST { '0000'B, '0001'B, '0010'B, '0011'B,
          '0100'B, '0101'B, '0110'B, '0111'B,
          '1000'B, '1001'B }}}
    REPETITION-ENCODING {
      REPETITION-SPACE
      -- We determine the concatenation of the bitstrings for the
      -- characters and add a terminator (e.g.,
      -- '0100'B + '0010'B becomes '0100 0010 1111'B)
        SIZE variable-with-determinant
        DETERMINED BY pattern
      PATTERN bits:'1111'B}}
```

Table 2.12: An ECN encoding object assignment for application on the ASN.1 assignment shown in 2.11, taken from [IT02c].

videoconferencing [IT06], or from the Lightweight Directory Access Protocol Version 3(LDAPv3) [WHK97]. ECN has been used in order to redesign existing protocol standards to use ASN.1, such as for the Bluetooth Session Description Protocol (SDP) [LDT01] or the ISO/IEC 7816-4:2005 standard for communicating with integrated circuit cards (ICCs), also known as smart cards [ISO05b].

Discussion

ASN.1 and ECN allow a set of data formats to be defined through separately specifying their abstract and transport syntax using both languages, and allowing format-compliant software components to be generated. For Telecommunications, the combination of both languages has the interesting property that protocol messages defined in ASN.1 can be reused with different transport encodings on different transports.

- **Definitions and models:** The combined use of ASN.1 and ECN provides a model to define a data format of messages to be exchanged. As the predefined ASN.1 codecs strongly coin the resulting data formats for ASN.1 messages, general applicability of this approach depends on the universality of ECN to provide for arbitrary forms of transport syntaxes. While general applicability of ASN.1 and ECN is sometimes asserted and assumed in literature, it is not readily apparent due to the high complexity of both standards, and it has neither been proven nor argued for substantially, according to this author's knowledge.
- **Classification:** The combination of ASN & ECN is a *declarative*, *machine-processible* approach. While the approach is the subject of a highly complex and voluminous specification, *no formalised model for universal applicability* is given.
- **Descriptive capabilities:** As with other approaches, the combination of ASN.1 and ECN provides support for segmenting transcoded data and decoding primitive data with bit granularity. The approach can make use of ECNs limited support for at least arithmetic transformations of transcoded data through ECN's *#TRANSFORM encoding class*, although this discounts more complex processes required for compressed or encrypted data. Explicit support for the concatenation of fragmented data has not been encountered in the highly complex ECN specification.

2.4.2 Concrete Syntax Notation 1

While the previously presented ASN.1 and ECN approach provides means for defining a set of messages for a new protocol, separately describing the abstract and transport syntax of a message to these specifications is a complex task. When there is no immediate need to switch the transport syntax as ASN.1 and ECN allows to, then directly specifying the representation of a message in bits and bytes is a simpler and more transparent approach.

Overview

The Concrete Syntax Notation 1 (CSN.1) specification allows the composition of a message from bits and bytes to be directly specified. CSN.1 itself has been specified by the *European Telecommunication Standards Institute (ETSI)* and is used in mobile communication standards of the *3rd Generation Partnership Project (3GPP)* [ETS10].

CSN.1 itself is quite similar to the *Extended Backus Naur Form (EBNF)*. In CSN.1, `0`, `1` and `null` are all terminals which either refer to bits of the respective value, or to the empty bitstring. An *expression* can be either a terminal, a *concatenation* of multiple expressions into a sequence, a *choice* of alternative expressions, or a *reference* to a rule name defined for an expression. An expression may be followed by parantheses denoting *repetition*, either by a fixed number through an integer value, or by a variable, unbound number through the Kleene star symbol "*". An expression itself can be tagged with a label for later identification of parts of a rule. Further extensions are introduced ad-hoc in specifications, such as defining a constraint on the value of an expression through the == operator. An example CSN.1 definition from 3GPP is shown in Table 2.13, where a so-called *Uplink RLC/MAC control message* type is defined as a 6-bit header identifying a message subtype, followed by the contents of the specific subtype itself. In this case, a 6-bit header with a value of `000000` identifies the following *Packet Cell Change Failure message content* type.

Discussion

CSN.1 can be used for a low-level description of bit-oriented messages in Telecommunication protocol for purposes of standardisation and possibly validation.

- **Definitions and models:** The CSN.1 specification provides a very simple approach towards data format description that is on the other end of the scale compared to ASN.1 and ECN. Basically, CSN.1 is similar to a form of EBNF specifially adapted for describing bit-oriented messages, so it provides means for describing data structures, but does not provide the means such as decoding BCD integers or similar.

- **Classification:** CSN.1 is a *declarative*, *machine-processible* approach, yet *no formalized model for universal applicability* is presented in literature.

- **Descriptive capabilities:** CSN.1 provides support for segmenting structured data and partial support for decoding primitive data, both with bit granularity. Lacking concepts for actual decoding of primitive data, CSN.1 can only be considered to provide an identity decoding of bit sequences. For example, if a bit sequence represented a signed integer, then decoding the bit sequence into an actual signed integer value is not within the capabilities of CSN.1 itself. Regarding the transformation of transcoded data or the concatenation of fragmented data, the CSN.1 approach does not provide support for either descriptive capabiliy.

```
< Uplink RLC/MAC control message > ::=
  < MESSAGE_TYPE : bit (6) == 000000 >
    < Packet Cell Change Failure message content > |
  < MESSAGE_TYPE : bit (6) == 000001 >
    < Packet Control Acknowledgement message content > |
  < MESSAGE_TYPE : bit (6) == 000010 >
    < Packet Downlink Ack/Nack message content > |
  < MESSAGE_TYPE : bit (6) == 000011 >
    < Packet Uplink Dummy Control Block message content > |
  < MESSAGE_TYPE : bit (6) == 000100 >
    < Packet Measurement Report message content > |
  < MESSAGE_TYPE : bit (6) == 001010 >
    < Packet Enhanced Measurement Report message content > |
  < MESSAGE_TYPE : bit (6) == 000101 >
    < Packet Resource Request message content > |
  < MESSAGE_TYPE : bit (6) == 000110 >
    < Packet Mobile TBF Status message content > |
  < MESSAGE_TYPE : bit (6) == 000111 >
    < Packet PSI Status message content > |
  < MESSAGE_TYPE : bit (6) == 001000 >
    < EGPRS Packet Downlink Ack/Nack message content > |
  < MESSAGE_TYPE : bit (6) == 001001 >
    < Packet Pause message content > |
  < MESSAGE_TYPE : bit (6) == 001011 >
    < Additional MS Radio Access Capabilities message content>;
```

Table 2.13: Excerpt of a 3GPP message definition in CSN.1 from [ETS05].

	XCEL	DFDL	MPEG 1/2	Flavor	BSDL	(g)BFlavor	ASN.1 & ECN	CSN.1
Machine-processible approach	☒	☒	☐	☒	☒	☒	☒	☒
Procedural approach	☐	☐	☒	☒	☐	☒	☐	☐
Declarative approach	☒	☒	☐	☐	☒	☐	☒	☒
Formalised model for universal applicability	☐	☐	☐	☐	☐	☐	☐	☐
Decoding of primitive data	⧄	☒	☒	☒	☒	☒	☒	⧄
Segmentation of structured data	☒	☒	☒	☒	☒	☒	☒	☒
Transformation of transcoded data	☒	☐	☐	☐	☐	☐	⧄	☐
Concatenation of fragmented data	⧄	☐	☐	☐	☐	☐	☐	☐

Table 2.14: Comparison of related work in terms of supported elementary descriptive capabilities.

2.5 Discussion

This discussion of the current State of the Art presents a *consideration of examined related work*, a number of *general observations* and references to *other approaches* that have not been addressed in this thesis, but which may be of interest for future research.

2.5.1 Consideration of examined Related Work

Using the classification and descriptive capabilities of models in related work, it is possible to compare approaches that have been surveyed by examining their support for decoding, segmentation, transformation and concatenation:

- **Classification:** Examined data format registries either retain natural-language descriptions or depend on formal descriptions given in one of the existing approaches. The OAIS Reference Model introduces the terminology of Representation Information for some representation of data format knowledge, yet primarily considers and thus depends on existing software implementations. A similar situation is with TOM, which allows the migration of data between data formats to be managed. It depends on external software implementations to provide actual conversion services, and therefore does not provide a model for making data format rules and constraints explicit.

  Formal models for describing data formats have been observed in XCEL, DFDL, the MPEG-1/2 methodology, Flavor, BSDL, (g)BFlavor, ASN.1 in combination with ECN and CSN.1. Nearly all approaches are machine-processible, with only the MPEG-1/2 methodology being an exception, as it employs a tabular notation intended for conveying data format knowledge to human engineers. The majority of approaches are declarative, as only the MPEG-1/2 methodology, Flavor and (g)BFlavor depend on procedural descriptions in the form of (pseudo) source code.

Yet, among all these approaches, no formalised model is presented in literature that systematically considers their universal applicability, or their inherent limitations. While every examined approach addresses specific use cases in their domains and while they have been put to use, the basic question of *sufficiency and necessity of descriptive capabilities* has not been investigated systematically, be it for data formats from a specific domain or in general.

There is a lack of a suitable formalised abstraction on data formats in general which could be used to establish *elementary descriptive capabilities for universal applicability.* These aspects are therefore addressed in the upcoming analysis of this thesis.

- **Descriptive capabilities:** The lack of a formalised model geared towards universal applicability also manifests itself in a lack of systematic support of descriptive capabilities, although authors at times claim universal applicability of their approach for describing arbitrary data formats. Yet, this occurs without giving some sort of proof, arguing the case substantially or at least making such expressive powers readily apparent for the reader.

  Unsurprisingly, decoding primitive data and segmenting structured data can be considered as minimally required descriptive capabilities for any approach addressing the description of nontrivial formats, and is thus supported by all approaches, with the minor exception of CSN.1 not handling the actual coding of primitive data, and with XCEL not handling data on bit granularity, but on octet granularity. Where approaches address the coding primitive data, but may not provide for their set of encodings to be extended, performing a specific extension would be trivial, but nevertheless lacks methodical consideration for extending encodings in general.

  Providing support for the transformation of transcoded data is more difficult, though. It both requires the ability to transform data and to re-process the result of the transformation, which is conceptually provided only by XCEL and only in part by ASN.1 & ECN. Like with encodings, extending their set of transformations, for example when compression or encryption algorithms are used in a data format, has not been considered in a thorough fashion.

  Support for the concatenation of fragmented data is even more difficult to provide, as it requires fragments to be concatenated in the right order, and to be able to re-process the resulting concatenation. From all approaches, only XCEL itself provides partial support for concatenating fragmented data by using the reserved *normData* symbol name, as shown for a PNG IDAT chunk sample [SHC08], and enabling re-processing of concatenated data by using the *internalSource* attribute. Yet active control of fragment ordering is not explicit, casting its universal applicability into doubt for potential data formats requiring alternative fragment orderings.

From a wide number of approaches on data format description, only the XCEL approach comes close to supporting all four descriptive capabilities used for comparison. Since XCEL is used to monitor the retention of significant information for real-life data migration projects, XCEL's near-complete support can probably be attributed to necessity-driven progress, yet comes at the price of a complex specification with nontrivial interactions of concepts.

2.5.2 General observations

In the course of the survey, a number of noteworthy observations have been made:

- **Distinction between describing and prescribing:** During the survey of literature, it became noticeable that publications at times fail to properly distinguish between *describing a data format* and *prescribing a data format*, although both goals are conceptually different. Prescribing a data format can content itself with providing a specific representation for a given message that suits a specific purpose, but this does not necessarily require it to be capable of producing arbitrary representations. For example, considering ASN.1 without ECN is clearly a prescriptive approach. On the other hand, describing a data format, unless explicitly restricted to a subset of data formats, has to handle the problem of arbitrary representations that may exist and be used.

- **Similarities between approaches:** Existing research in literature provides approaches for domain-specific use cases that fit well, such as BSDL for high-level content adaptation of scalable formats in Multimedia, or ASN.1 for ensuring interoperability between sender and receiver in Telecommunications.

  Several approaches have developed along similar concepts and lines of thought in different domains. For example, the idea of extending XML Schema can be found in both DFDL from Digital Preservation and BSDL from Multimedia, both using XML Schema annotations for their specific extensions.

  Yet, cross-pollination of approaches has basically been confined to within a domain, such as with Flavor, BSDL and their subsequent recombination in BFlavor and gBFlavor.

- **Necessity of round-trip support:** Data formats define representations of information to be exchanged. For a proper exchange of information between sender and receiver, an unambigious mapping between information and its representation is required, supported by the necessity of round-trip support for parsing and serialisation as explicitly required by DFDL. The need for an unambigious mapping influences the central research hypothesis in the upcoming analysis of this thesis.

- **Tight coupling of data format knowledge and format-compliant data:** Format-compliant data and the format-compliant knowledge embedded in applications are tightly coupled, as it is this knowledge in applications which assigns semantics to information stored using data formats, and which is required to access the contained information. This becomes especially visible in Digital Preservation regarding the *migration of data*, trying to retain as much information as possible for other applications to provide access to, or the *emulation of applications* so they can still be used for access.

- **Varying granularity of description:** Depending on the actual purpose of describing data, such as for high-level content adaptation, or for obtaining a specific information, suitable descriptions may have different degrees of granularity. A typical argument in favour of a limited, coarse granularity of description is that a more fine-grained description would be too verbose, as data formats typically provide a space-efficient representation for information. While a description can still be sufficient at a coarse granularity, arbitrary degrees of granularity down to the level of bits may be required in order to resolve and handle dependencies in data (such as the high-level context problem introduced by Flavor).

- **Different kinds of data in a description:** As seen from DFDL's "hiding" of data contained in binary representations, there are two different roles of data present. One role is data that carries original information, which is actually of interest to a user and that is associated with a given representation. The other role is data that "just" serves for wrapping up and transporting the original information, such as length information or description of data types.

- **Support for partial descriptions of format-compliant data:** Partial descriptions of format-compliant data, such as shown by XCEL, can be helpful in case of incomplete data format knowledge, or when a description is still under construction.

2.5.3 Other approaches

The set of works presented here is a selection of essential and influential approaches from the data-centric domains of Digital Preservation, Multimedia and Telecommunication.

Naturally, further approaches remain that have not been selected for presentation in this thesis, and whose in-depth examination, including their relations to other approaches, is left for future research. These approaches include, but are not limited to:

- the *Binary XML (BinX)* description language [Wes02] and its relation to DFDL, for which BinX was a precursor effort,

- the *Binary Format Description (BFD)* [MC03], another approach using XML representations of binary data for sharing of scientific data,

- the *Enhanced Ada SubseT (EAST)* data description language, addressing needs in data description for space-related information [CCS00],

- the *Abstract Syntax Description Language (ASDL)* [WAKC97], which is concerned with intermediate representations of programming languages,

- the PacketTypes packet specification language [MC00] to allow the specification of network protocol messages through types,

- the DataScript specification and scripting language for binary data, with similarities to C and Java "tapping into programmers' existing skill sets" [Bac02],

- the Hancock language for the analysis of "transactional data streams" for purposes of data mining [CFP+04], and
- the PADS language intending to support programmers for parsing "ad-hoc data" following a given data format [FG05], including a calculus of "dependent types" for data description languages [FMW06].

Two perhaps known approaches have been deliberately excluded from presentation, as they lacked both the intention and capability to describe arbitrary data formats, or failed to have a notable impact on the current State of the Art. These are the *External Data Representation (XDR)* and *Transfer Syntax Notation One (TSN.1)* languages:

- Although the *External Data Representation (XDR)* standard is used for describing the composition of PDUs used in the *Network File System (NFS)* protocol, XDR explicitly does not intend to describe arbitrary data formats. For example, XDR has no support for bitstreams of non-octet length.
- The *Transfer Syntax Notation One (TSN.1)* is a data description language used in a commercial product that resembles a cross of CSN.1 and Flavor stripped from its object-oriented aspects. Yet, it has neither been found to be actively used in publically available specifications, nor does it provide readily apparent means for concatenating fragmented data or processing data stored in a compressed or encrypted form.

2.6 Summary

This survey on current State of the Art in data format description has shown that from Digital Preservation, Multimedia and Telecommunication, data format description plays a role in a multitude of use cases, where the primary focus is on the migration of data in Digital Preservation, and the normative definition of data formats for files in Multimedia and for protocol data units in Telecommunication.

For these use cases, a variety of related research, such as data format registries or the OAIS reference model, and a number of approaches from related work have been identified, which often provide a domain-specific viewpoint. Definitions and models from related work have varying properties and differ in expressiveness, yet sometimes share common approaches even across research domains, such as the idea of extending XML Schema for use on binary data. General applicability for describing arbitrary data formats is sometimes assumed, yet, neither a formalisation on data formats in general nor a proof on general applicability of an approach to data format description has been given.

Chapter 3

Analysis

3.1 Introduction

The previous Chapter 2 surveyed the current State of the Art in literature regarding data formats in various domains of research, including several contributions to describing the composition of data. While several approaches from related work claim general applicability for arbitrary data formats, these do not substantiate their claims on a theoretical level. In examined literature, describing data formats in general has neither been subject to systematic investigation, nor have inherent properties of data formats and potential limits of describing the composition of data been considered in-depth.

This chapter therefore analyses inherent properties in data format instances and data formats in general, specifically addressing the following questions:

- **What is a formalised abstraction of a data format instance and a data format, which is suitable for universal applicability?** The abstraction provides a basis for discussing inherent properties and problems of as well as limits to data format description on a theoretical basis.

- **What are elementary descriptive capabilities required for universal applicability?** Exploring the formalised abstraction leads a set of elemental descriptive capabilities that are required for describing arbitrary data formats through their data format instances.

- **What are limits to data format description?** Given a formalised abstraction and a set of required descriptive capabilities for general applicability, exploring their limits shows what can be reasonably expected from data format descriptions.

This chapter introduces a research hypothesis for analysis in Section 3.2 and builds a formalisation of data format description in Section 3.3. Its properties are considered in Section 3.4, and inherent limits on data format description in general are examined in 3.5. Finally, the chapter closes with a discussion and a summary.

3.2 Research Hypothesis

The OAIS Reference Model introduced in Chapter 2.2.2 states that a data format is “representation information”, which “maps a set of bit sequences into more meaningful concepts”. Due to the necessity of round-trip support for parsing / decoding and encoding / serialisation observed in Chapter 2.5.2, such a mapping has to be unambigious, therefore bijective and thus lossless. For this analysis, the following research hypothesis describes the notion of a data format more closely:

Hypothesis 3.2.1: The current State of the Art regarding data format description can be improved by assuming that a data format defines a normative set of lossless information representations, passed as messages between a sender and a receiver for the purpose of storage and transmission over time.

$$\begin{tikzcd}
\text{sender } \alpha & \text{channel } c & \text{receiver } \beta & \\
m_\alpha \arrow[d, "{v_d^\alpha,\ f_{\alpha\to d}}"'] & & m'_\beta & \text{internal} \\
m_d \arrow[rr, "f_c"] & & m'_d \arrow[u, "{v_d,\ f^{-1}_{\beta\to d}}"'] & \text{external}
\end{tikzcd}$$

Figure 3.1: Abstraction of the information transport from a sender α to a receiver β using a data format d

Let $\mathbb{D}$ denote the set of all data formats, $d \in \mathbb{D}$ denote a data format, α denote a sender and β denote a receiver. Following the hypothesis, the basic usage scenario of a data format shown in Figure 3.1 can be stated as follows:

- A sender α has an “internal representation” m_α of information. The sender ensures the validity of m_α with respect to a data format d and maps from m_α to an “external representation” m_d, which is then sent over a channel c.
- A receiver β eventually obtains an “external representation” m'_d from c, which may be different from the sent m_d in case of a noisy channel, or invalid due to an erroneous sender. The receiver therefore ensures the validity of m'_d and maps from m'_d to an “internal representation” m_β.

Sender and receiver necessarily have to share information regarding the data format d. Moreover, depending on d, both α and β may share additional context information required for deciding about the validity, or for mapping from and to an external representation m_d. Two examples of such context information are the use of encryption in a data format, and the identification of embedded data formats through separate channels.

3.3 Formalising Data Format Description

Informally, we define a *data format instance* as a mapping between an internal m_γ and an external representation m_d, and define a *data format* through a set of such instances. Formalising both terms, the following definitions are given step-wise as follows.

3.3.1 Representing primitive information

The formalisation begins with the most basic element, a *bit sequence*:

Definition 3.3.1 (Bit Sequence): A bit sequence b is defined as finite and non-empty. The set of all finite, non-empty bit sequences is defined as $\mathbb{B}$ (Eq. 3.1).

$$b = \{0,1\}^n, n \geq 1, b \in \mathbb{B} \tag{3.1}$$

Example 3.3.1: Let b_1 be the finite and non-empty bit sequence 11100000.

Definition 3.3.2 (Encoding): An *encoding* e is a bijective function which maps between an element $x \in \mathbb{X}$ from some arbitrary domain $\mathbb{X}$ and its corresponding bit sequence b (Eq. 3.2).

$$e : \mathbb{B}_e \leftrightarrow \mathbb{X}_e, \mathbb{B}_e \subseteq \mathbb{B}, \mathbb{X}_e \subseteq \mathbb{X} \tag{3.2}$$

Example 3.3.2: Let e_{LSBI}, e_{ASCII} be encodings, where e_{LSBI} denotes a least-significant bit first integer encoding, and e_{ASCII} denotes an ASCII character string encoding. In this case, $e_{LSBI}(7) = 11100000$, which equals b_1 from the previous Example 3.3.1, and $e_{ASCII}(\text{“}AB'') = 10000010\,01000010$.

A bit sequence represents encoded *data*, but does not describe its meaning by itself, as it depends on the actual context. In order to represent data including its semantics as information, some sort of "labeling" is needed.

Definition 3.3.3 (Label): A *label* l is a symbol that denotes some given semantics. The set of all labels is defined as $\mathbb{L}$.

Definition 3.3.4 (Labeled Bit Sequence): A *labeled bit sequence* i is defined as a pair $i = (b, \mathbb{L}_i)$, where $b \in \mathbb{B}$ is a bit sequence and $\mathbb{L}_i \subseteq \mathbb{L}$ is a subset of labels that denote the meaning of b (Eq. 3.3). The set of all labeled bit sequences is defined as $\mathbb{I}$.

$$i = (b, \mathbb{L}_i), b \in \mathbb{B}, \mathbb{L}_i \subseteq \mathbb{L}, i \in \mathbb{I} \tag{3.3}$$

Example 3.3.3: Let ω, θ be labels, where ω denotes the meaning "Intel x86 machine opcodes" and θ denotes the meaning "24bit RGB colour triplet", and let $b = 10010000\,10010000\,10010000$. In this case, $(b, \{\omega\})$ represents the Intel x86 opcode sequence `NOP NOP NOP` consisting of three "do nothing" machine instructions. At the same time, $(b, \{\theta\})$ represents the RGB colour triplet `#909090`, corresponding to a dark gray color.

A labeled bit sequence represents *information* by making the meaning of encoded data as a bit sequence explicit for a specific context. It can be categorised as either *payload* or *packaging*, depending on whether its information is part of the message to be transported, or whether it is used for transportation:

Definition 3.3.5 (Payload): A labeled bit sequence $i = (b, \mathbb{L}_i)$ is *payload* if the value of its bit sequence b is functionally independent from other labeled bit sequences.

Definition 3.3.6 (Packaging): A labeled bit sequence $i = (b, \mathbb{L}_i)$ is *packaging* if the value of its bit sequence b is functionally dependent on one or more labeled bit sequences, such as depending on their (relative) location, length, labels or bit sequences.

Example 3.3.4: Let ω, θ be labels, where ω denotes the meaning "text comment" and θ denotes the meaning "length of the text comment". Let $b_1 = e_{ASCII}(\text{“}comment''))$ and $b_2 = e_{LSBI}(7)$. In this case, $(b_2, \{\theta\})$ is packaging, as it has a functional dependency on $(b_1, \{\omega\})$. $(b_1, \{\omega\})$ itself is functionally independent and thus payload.

Example 3.3.5: For example, given the PNG image file format which may carry a string of text in a "tEXt" chunk data structure; in this case, the *keyword* element contained in the *chunk data* element of the data structure is a functionally independent text string and therefore a payload element. On the other hand, both the *chunk length* and the *CRC* element are functionally dependent and thus represent packaging elements.

3.3.2 Representing complex information

Labeled bit sequences serve as building blocks for more complex representations, which are used either as *internal representation* at a sender or receiver γ, or used as *external representation* for exchanging information.

Definition 3.3.7 (Internal Representation): An *internal representation* m_γ represents information in a way that is specific to some sender / receiver γ and is defined as a tuple of one or more labeled bit sequences (Eq. 3.4) which have defined semantics. The set of all possible internal representations is defined as $\mathbb{IR}$.

$$m_\gamma = \{i_1, \ldots, i_n\}, n \geq 1, i_x \in \mathbb{I}, m_\gamma \in \mathbb{IR} \tag{3.4}$$

Different internal representations may represent the same information, yet in varying *granularity.*

Definition 3.3.8 (Granularity): The *granularity* of an internal representation m_γ is a relative measure on how fine-grained information is represented. Finer granularity is achieved by a more fine-grained description. The actual granularity of m_γ may vary depending on the processing needs of sender or receiver γ.

Example 3.3.6: Let $m_{\gamma,1} = \{i\}, i = (b, \mathbb{L}_i)$ be an internal representation, where i represents the colour of a pixel as a 24 bit RGB value composed of 8 bits for each colour component of red, green and blue. Let $m_{\gamma,2} = \{i_1, i_2, i_3\}, i_x = \{b_x, \mathbb{L}_{i,x}\}$ be an internal representation, where i_1, i_2 and i_3 represent the colour of a pixel as a 24 bit RGB value as separate 8 bit red, green and blue colour components. In this case, $m_{\gamma,2}$ has a finer granularity than $m_{\gamma,1}$.

Packaging is typically present in a data format in order to describe variable aspects of payload required during the parsing process, such as the length of a variable-length payload. In order to compute packaging information during generation and to process packaging information during parsing, a certain minimum granularity of internal representation is required which separates packaging from payload.

DEFINITION 3.3.9 (EXTERNAL REPRESENTATION): An *external representation* m_d represents information as normatively defined by a data format d. It is defined as a tuple containing exactly one labeled bit sequence (Eq. 3.5). The set of all possible external representations is defined as $\mathbb{ER}$.

$$m_d = \{i\}, i \in \mathbb{I}, m_d \in \mathbb{ER} \tag{3.5}$$

EXAMPLE 3.3.7: Let ω be a label which denotes the meaning "Portable Network Graphics (PNG) image file". Let b_{PNG} denote the bit sequence of a valid PNG image file, and let i_{PNG} be the labeled bit sequence $\{b_{PNG}, \{\omega\}\}$. In that case, $m_{d_{PNG}} = i_{PNG}$ is an external representation of an image in the Portable Network Graphics image file format.

An external representation m_d typically carries some aggregation of information rather than a single primitive value. Such a case is shown in Example 3.3.7, where the external representation $m_{d_{PNG}}$ carries an aggregation of information, which contains the width and height of the actual image as primitive values among others.

3.3.3 Validating representations

Given some internal representation m_γ or external representation m_d, it is necessary to test their validity through *validation functions*:

DEFINITION 3.3.10 (INTERNAL VALIDATION FUNCTION): For a given sender α and data format d, an *internal validation function* is denoted as v_d^α (Eq. 3.6). An internal representation m_α is *valid* iff $v_d^\alpha(m_\alpha) = 1$. The subset of all valid internal representations of α for d is defined as $\mathbb{IR}_d^\alpha \subseteq \mathbb{IR}$.

$$v_d^\alpha : \mathbb{IR} \to \{0,1\} \tag{3.6}$$

EXAMPLE 3.3.8: Let α be a sender of PNG images with random pixel data, where the user determines the resolution by entering its width and height on the keyboard, and let the user enter a width of 0 and a height of 1, which is passed on as image width and height in its internal representation. In this case, the internal representation is invalid, as the PNG image file format specifies the constraint that the image width may not be zero.

As can be seen in Example 3.3.8, data formats may be restricted to transporting specific types of information, where format constraints have to be met. A sender α therefore must at least be able to test whether an internal representation m_α is valid according to d or not.

DEFINITION 3.3.11 (EXTERNAL VALIDATION FUNCTION): For a given $d \in \mathbb{D}$, an *external validation function* is denoted as v_d (Eq. 3.7). An external representation m_d is *valid* iff $v_d(m_d) = 1$. The subset of all valid external representations for d is defined as $\mathbb{ER}_d \subseteq \mathbb{ER}$.

$$v_d : \mathbb{ER} \to \{0,1\} \tag{3.7}$$

Example 3.3.9: Let β be a receiver of PNG images, where bit errors during transport have set the image width to zero. In this case, the external representation is invalid, as the PNG image width may not be zero, just as in the previous Example 3.3.8.

A received external representation m'_d may be invalid, for example due to a degrading storage medium or due to interference on a network link. Therefore, a receiver β must at least be able to test whether the received m'_d is valid.

In order to transport information from the internal representation $m_\alpha \in \mathbb{IR}_d^\alpha$ to the external representation $m_d \in \mathbb{ER}_d$, and vice versa from a valid external representation $m'_d \in \mathbb{ER}_d$ to the internal representation $m'_\beta \in \mathbb{IR}_d^\beta$, a suited mapping between both sets becomes necessary.

3.3.4 Mapping between representations

Hypothesis 3.2.1 states that information representation is lossless. When considering the internal representation as the *information to be represented*, and the external representation as the *information representation*, the hypothesis leads to the requirement that the mapping between internal and external representations is lossless and thus bijective. It is now necessary to consider the mapping between internal and external representations through *mapping functions*:

Definition 3.3.12 (Mapping function): For a given sender α and data format $d \in \mathbb{D}$, a bijective *mapping function* $f_{\alpha\to d}$ (Eq. 3.8) maps from $\mathbb{IR}_d^\alpha$ to $\mathbb{ER}_d$ through *encoding* and *serialisation*. For a given receiver β and data format $d \in \mathbb{D}$, its inverse $f^{-1}_{\beta\to d}$ (Eq. 3.9) maps from $\mathbb{ER}_d$ to $\mathbb{IR}_d^\beta$ through *parsing* and *decoding*.

$$f_{\alpha\to d} : \mathbb{IR}_d^\alpha \to \mathbb{ER}_d \tag{3.8}$$
$$f^{-1}_{\beta\to d} : \mathbb{ER}_d \to \mathbb{IR}_d^\beta \tag{3.9}$$

For a given d and α, due to the required bijectivity of mapping functions, both sets $\mathbb{ER}_d$ and $\mathbb{IR}_d^\alpha$ necessarily have the same size - for every external representation m_d, there exists a corresponding m_α, and vice versa.

Example 3.3.10: Let d be the MPEG-2 Transport Stream (MPEG-2 TS) data format [ISO00], which serves to stream packetised video, audio and auxillary data over lossy channels, and which is for example used for digital television to be carried over satellite or cable. Without an explicit upper bound to the number of packets in an MPEG-2 TS, and thus without an upper bound to the bit sequence length of its external representations, there is an infinite number of external representations $m_d \in \mathbb{ER}_d$.

Depending on whether a data format d has a maximum bit sequence length for its external representations, the sets $\mathbb{ER}_d$ and $\mathbb{IR}_d^\alpha$ may be finite. Both sets $\mathbb{ER}$ and $\mathbb{IR}$ are infinite.

Given an external representation m_d to be exchanged between sender and receiver, the notion of a *channel* is required:

DEFINITION 3.3.13 (CHANNEL): A *channel* c passes an external representation $m_d = \{i\}, i = \{b, \mathbb{L}_i\}$ from a sender α to a receiver β, including the bit sequence b and its labels $\mathbb{L}_i$. It is modelled as a channel function f_c (Eq. 3.10).

$$f_c : \mathbb{ER} \to \mathbb{ER} \tag{3.10}$$

EXAMPLE 3.3.11: Let ω be a label denoting a PNG image file. Let $f_{c,1}$ be a channel representing a file in the File Allocation Table 32 (FAT32) file system. For a given external representation $m_d = i, i = \{b, \{\omega\}\}$ of a PNG image file, the bit sequence b is stored as file content in the FAT32 file system, while its label ω is stored using the *file name extension* ".png".

EXAMPLE 3.3.12: Let ω be a label denoting an MPEG-2 Audio stream. Let $f_{c,2}$ be one of multiple channels provided by an MPEG-2 Transport Stream. For a given external representation $m_d = i, i = \{b, \{\omega\}\}$ of an MPEG-2 Audio stream, the bit sequence b is interleaved in the MPEG-2 Transport stream, while its label ω is encoded in a Program Map Table (PMT), which refers to the MPEG-2 Audio data format.

A channel c handles the transmission of an external representation $m_d = \{i\}, i = \{b, \mathbb{L}_i\}$ by transferring both the bit sequence b and its labels $\mathbb{L}_i$. A specialised channel c may only pass external representations for a specific set of data formats. Furthermore, a channel c may be noisy and introduce errors into the bit sequence or the set of labels.

In order to map between internal and external representations, some means for a bijective *mapping step* is needed.

DEFINITION 3.3.14 (MAPPING STEP): A *mapping step* t is a bijective function which maps between *input* and *output* as two ordered tuples of bit sequences (Eq. 3.11).

$$t : \mathbb{B}^n \leftrightarrow \mathbb{B}^m, n \geq 1, m \geq 1 \tag{3.11}$$

As shown in Figure 3.2, mapping steps can be categorised through the cardinality of the input and output tuples as

- a *segmentation* of structured data $(1 : m)$,
- a *transformation* of transcoded data $(1 : 1)$, or
- a *concatenation* of fragmented data $(n : 1)$, forming a composite.

Arbitrary $n : m$ mapping steps can be composed from segmentations, transformations and concatenations. A mapping step may optionally use additional parameters that control the bijective mapping.

EXAMPLE 3.3.13: Let t be a *Run-Length Encoding* (RLE) block transformation, where a sequence of n equivalent bits is considered a *run* with a maximum length of 4, and where the length n is encoded as $n - 1$ using 2 bits, followed by the bit to be repeated. Given a bit sequence $b = 00001111$, then b consists of two runs of length 4, where the first run is composed from 0s, and the second run is composed from 1s. Therefore, $t(b) = 110111$.

$$\begin{tikzcd}
(b_0) \arrow[d] & (b_0) \arrow[d] & (b_0,\dots,b_m) \arrow[d] \\
(b_1,\dots,b_{n+1}) & (b_1) & (b_{m+1}) \\
\text{segmentation} & \text{transformation} & \text{concatenation}
\end{tikzcd}$$

Figure 3.2: Mapping steps ordered by input and output cardinality.

$$\begin{tikzcd}
m_d = \{i\} \arrow[d, bend left] \arrow[dddd, bend left=60, "f^{-1}_{\alpha\to d}(m_d)"] \\
\dots \arrow[u, bend left] \arrow[d, bend left] \\
\{i_1,\dots,i_m\}, m \geq 1 \arrow[u, bend left] \arrow[d, bend left] \\
\dots \arrow[u, bend left] \arrow[d, bend left] \\
m_\alpha = \{i_1,\dots,i_n\}, n \geq 1 \arrow[u, bend left] \arrow[uuuu, bend left=60, "f_{\alpha\to d}(m_\alpha)"]
\end{tikzcd}$$

Figure 3.3: Bijective mapping between m_α and m_d

The bijective mapping between an internal representation m_γ and an external representation m_d as defined through transformations gives rise to a *data format instance.*

Definition 3.3.15 (Data format instance): Given a pair of representations (m_d, m_γ) with $m_d = \{i_0\}, m_\gamma = \{i_1,\dots,i_n\}, n \geq 1$, a *data format instance* is a rooted, directed, ordered, acyclic graph as a *causality graph* on labeled bit sequences. The graph is rooted in i_0 and has $i_1,\dots,i_n$ as its leaves. The graph is composed from a finite set of mapping steps, where each mapping step t defines directed arcs from an ordered set of input bitstream segments to an ordered set of output bit sequences. Regarding intermittent nodes in the causality network, their bit sequences are the result of mapping steps, while their labels functionally depend on neighbouring labeled bit sequences as well as on optional context information depending on d.

When considering fragmented data for concatenation, the order of fragments is represented in the order of input bitstream segments mapping the fragments to a single output bitstream segment as their (ordered) concatenation. Intermittent nodes carry intermediate values resulting from the mapping steps between the internal and external representation.

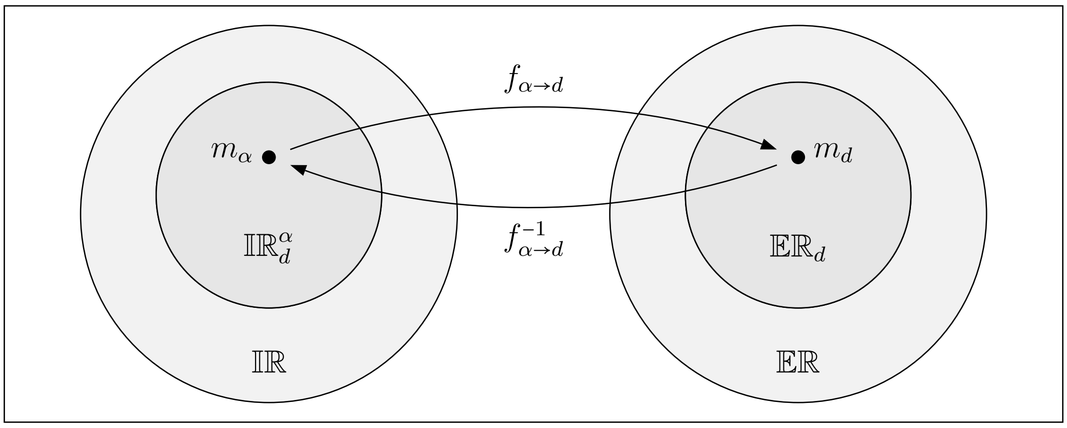


Figure 3.4: Bijective mapping between internal representations $\mathbb{IR}^\alpha_d$ and external representations $\mathbb{ER}_d$ through mapping function $f_{\alpha\to d}$ and its inverse $f^{-1}_{\alpha\to d}$

Example 3.3.14: The file signature of a valid PNG image file obtains its semantics from being the first element in its bit sequence, having a fixed length of 8 bytes and a defined value. On the other hand, an *IHDR chunk* data structure in the same file obtains its semantics from the value of the second element, which is located 4 bytes after its start, has a length of 4 bytes and contains the ASCII string value "IHDR".

The causality graph of a data format instance can be compared to a map that locates its specific elements and how they come into place. As a data format instance belongs to a specific external representation, on this level, there is no consideration of "choice" or "alternatives", as one might have expected.

Building on previous definitions, the conceptual notion of a *data format* can now be defined for analysis:

Definition 3.3.16 (Data format): A data format d is a potentially infinite set of data format instances, which maps between a normative $\mathbb{ER}_d \subseteq \mathbb{ER}$ and a canonical $\mathbb{IR}^\gamma_d \subseteq \mathbb{IR}$, which is intended for transmission over a channel c.

3.4 Properties of the Formalisation

This section starts by considering the *suitability of bijective mapping functions for data format mappings* as well as the *sufficiency for lossless and lossy data formats* for the presented formalisation to counter potential misconceptions, and gives insights into its *sufficiency and necessity of descriptive capabilities.*

3.4.1 Suitability of bijective mapping functions for data formats

The presented formalisation requires the bijectivity of mapping functions, which is not suggested by most existing data format descriptions. This stands out especially when there are multiple external representations $m_{d,1}, \ldots, m_{d,n}$ that are seemingly equivalent representations for the same internal representation m_γ. Assuming this seeming equivalence, this may lead to the misconception that the bijective mapping requirement is overly restrictive, which requires that there is a one-to-one correspondence between internal and external representations.

Following the requirement of bijective mapping functions, the existance of multiple seemingly equivalent external representations can be considered as the result of a side-channel carrying information, which has been neglected in the internal representation m_γ, but which actually distinguishes between the external representations $m_{d,1}, \ldots, m_{d,n}$. Such differences in external representations can be detected in their bit sequences and reacted upon by a receiver, leading to potentially different behaviour. Therefore, neglected side-channels could lead to unintended side-effects, as shown in the example below:

Example 3.4.1: Let d be the Apple QuickTime data format, where media information is stored within a data structure called *MDAT atom*, and where the actual placement of individual media samples within the MDAT atom is stored within a data structure called *MOOV atom*. In typical Apple QuickTime movies, MDAT atoms are substantially larger than MOOV atoms, as the former is carrying the actual media samples, while the latter carries "just" related management information. Let $m_{d,1}$ be an Apple QuickTime movie where the MDAT atom is located prior to the MOOV atom in the bit sequence, and let $m_{d,2}$ be the very same Apple QuickTime movie, yet with the MOOV atom being located prior to the MODAT atom in the bit sequence. Assume both $m_{d,1}$ and $m_{d,2}$ to be located on a local harddisk, where both can be played back immediately using a suitable multimedia player such as Apple QuickTime or the VideoLan Client. In this context, $m_{d,1}$ and $m_{d,2}$ can be considered as seemingly equivalent despite different representations.

Now assume both $m_{d,1}$ and $m_{d,2}$ to be located on a remote web server, from which both movies have to be downloaded via HTTP for local playback. In the case of $m_{d,1}$, although the media samples in the MDAT atom are received first, their specific, individual placement in terms of start and length, as well as their type, still remains unknown. Playback of $m_{d,1}$ thus has to be delayed until the MOOV atom has been received as well. In the case of $m_{d,2}$, playback may start after the MOOV atom has been received, since the placement of individual media samples in the yet-to-come MDAT atom is now known to a multimedia media player. This property is termed "fast-start movie playback" by Apple.

In this context, the difference between $m_{d,1}$ and $m_{d,2}$ leads to a different playback behaviour that marks them as explicitly different in everyday use. If both external representations $m_{d,1}, m_{d,2}$ are considered seemingly equivalent to an internal representation m_γ, then m_γ is lacking the information whether the movie actually supports "fast-start movie playback".

With the requirement of bijective mapping functions and its enforcement, no such neglected side-channels can exist. The problem of seemingly equivalent multiple external representations $m_{d,1}, \ldots, m_{d,n}$ can be solved by extending the internal representation m_γ to $m_{\gamma,1}, \ldots, m_{\gamma,n}$ as necessary. Therefore, the requirement of bijective mapping function enforces the active consideration and handling of neglected side-channels in the design of data format mapping functions.

Moreover, the bijective mapping function requirement also enforces the clean separation of concerns related to the representation of information in a given data format, and the (typically lossy) conversion of information between different data formats. Such lossy, approximative conversion processes between different data formats, for example for converting text documents between the Microsoft Office Word 2010 .docx data format and the OpenDocument Format used by OpenOffice.org 3.2.1, are neither subject of the presented formalisation nor of this thesis.

3.4.2 Sufficiency for lossless and lossy data formats

Although a data format has been defined to specify the representation of information *in a lossless manner* by requiring bijective mapping functions, this does not limit the scope of the definition to lossless data formats only, as one might assume. A lossless data format represents original information, whereas a lossy data format represents the approximation of original information according to a defined metric, using some preprocessing function that filters information. In any case, the represented information is to be recoverable without loss by the receiver when assuming an error-free channel c, be it some original information or an approximation.

Therefore, mappings from and to m_d are required to be bijective and thus to be information-preserving. Aspects related to preprocessing of information according to some metric as well as postprocessing are not within the scope of this thesis.

EXAMPLE 3.4.2: Let d be the MPEG 1 Audio Layer 3 (MP3) audio file format, and m_d be an MP3 file. The data format d is a lossy data format, as it employs an approximation metric that is based on an acoustic perception model for humans. Although counterintuitive at first, m_d does not carry audio data itself, but merely a representation of an approximation of audio data that has been transported from some sender α. This representation can be recovered by a receiver β, completing the abstract information transport.

3.4.3 Sufficiency and necessity of descriptive capabilities

Based on the notions of encoding and transformation from Definitions 3.2 and 3.11, a set of *descriptive capabilities* that consists of *decoding, segmentation, transformation and concatenation* can be observed, which exactly match the descriptive capabilities used in the survey of related work in Chapter 2. These serve for handling *primitive data, structured data, transcoded data and fragmented data*, respectively:

- **Decoding of primitive data:** Being the most essential descriptive capability, it decodes a bit sequence representing a typed primitive to its domain-specific value and thus reverses the encoding operation of the sender. A simple example of primitive data is a bit sequence representing a *most significant bit first (MSBF)*-encoded integer representing the width in pixels of a PNG raster image.

- **Segmentation of structured data:** Since data formats rarely serve for representing single primitive values, it segments a structured bit sequence into its constituting parts, thereby reversing their concatenation performed by the sender. An example of structured data is a bit sequence containing the header of a *Portable Network Graphics (PNG)* raster image, the so-called *IHDR chunk* data structure, which has separate fields carrying information such as the width and height of the stored image.

- **Transformation of transcoded data:** Required for when a data format employs compression, encryption or another form of block transformation, it reversibly transforms an original bit sequence into a transcoded bit sequence, thereby reversing the block transformation that was applied to the transcoded bit sequence by the sender. An example of transcoded data is the PNG raster image data, which has undergone both a reversible scanline transformation as well as a lossless compression employed in PNG image files.
- **Concatenation of fragmented data:** For when a data format allows the data fragmentation, it concatenates multiple fragments into a composite, thereby reversing the fragmentation as performed by the sender. Examples of fragmented data are the fragmentation of transformed and compressed PNG raster image data in separate IDAT chunks allowed in PNG image files, as well as the time-based interleaving of audio and video data in multimedia containers such as the MPEG-4 File Format.

Regarding the question whether this is a set of *elemental descriptive capabilities* required for describing bitstreams from arbitrary data formats, assume a causality graph that describes the mapping between a bitstream as root node and a set of primitive values as leaf nodes. Representing the finite encoding and serialisation process performed by the sender, the graph must be finite as well. As to maintain causality of the encoding and serialisation process, there may not be loops within the graph, thus turning it into a causality graph. So for every node, there exists a finite *upward path* towards the root as well as one or more finite *downward paths* to primitive data. Paths in either upward or downward direction may be of zero length when the node in question is either the root node or a leaf node.

Exploring the causality graph from its root node in the downward direction, nodes in the causality graph are either leaf nodes or non-leaf nodes. In this case, every leaf node is a *decoded primitive*. Concerning non-leaf nodes, these participate either in a $1:m$ mapping designating the node to be a *segmented structure*, in a $1:1$ mapping designating the node to be a *transformed transcode*, or in a $n:1$ mapping designating the node to be a *concatenated fragment*, while $n:m$ mappings can be decomposed into the previous alternatives.

This set of descriptive capabilities is both sufficient and necessary for describing the composition of arbitrary data, as there are exactly these and no other types of elemental mappings in a causality graph besides the presented $1:m$, $1:1$, $n:1$ and $n:m$ non-leaf cases and the leaf case. Therefore, we consider this set as *elemental descriptive capabilities* required for universal applicability of an approach for data format description.

3.4.4 Using a PNG raster image as "litmus test"

To show that there are bitstreams from existing data formats that exercise the full set of *elemental descriptive capabilities*, readers are encouraged to consider the valid PNG raster image "oi2n0g16.png" from a PNG image test suite [vS98].

Let us assume to describe the composition of aforementioned PNG raster images in order to access colour information of a specific pixel. According to the PNG format description [Duc03] and the bitstream of the given file, segmentation is required for isolating two bit sequences contained in two *IDAT chunk* data structures. These bit sequences represent fragments of compressed, reordered image data that need to be concatenated first in order to decompress and then again reorder the image data, making it accessible. Finally, accessing individual pixel data requires segmentation of pixel and colour data, and decoding the stored primitive value. This PNG raster image can thus serve as a *litmus test* for data format description, and is thus revisited in later chapters.

3.5 Limits to Data Format Description

3.5.1 Overview

Inherent properties of data formats are present in the flow of information in Figure 3.1 and the bijective mapping in Figure 3.4, which can be stated as requirements for modelling data format instances and data formats as follows:

- **Validation of external representation:** The first two requirements ensure that a sender can distinguish between valid and invalid internal representations, and that a sender is capable to map an internal representation to an external representation by generation. Otherwise, a sender might create invalid external representations, or even loop trying to finish the generation process. In that case, no valid message is produced.

  REQUIREMENT 3.5.1: *The sender α can decide whether an internal representation m_α is valid using a function v_d^α.*

  REQUIREMENT 3.5.2: *The sender α can compute a valid external representation m_d from a valid internal representation m_α using a bijective mapping function $f_{\alpha \to d}$.*

- **Validation of internal representation:** The next two requirements ensure that a receiver can distinguish between valid and invalid external representations, and that a receiver is capable to map it to an internal representation by parsing. Otherwise, a receiver might create invalid internal representations, or even loop trying to finish the parsing process. In that case, no valid message is consumed.

  REQUIREMENT 3.5.3: *The receiver β can decide whether an external representation m'_d is valid using a function v_d.*

  REQUIREMENT 3.5.4: *The receiver β can compute a valid internal representation m'_β from a valid external representation m'_d using a bijective mapping function $f^{-1}_{\beta \to d}$.*

- **Bijective mapping between internal and external representations:** The last requirement ensures the unambiguousness and consistency of external and internal representations. For every external representation $m_d \in \mathbb{ER}_d$, there exists exactly one corresponding internal representation m_γ. As well, for every internal representation $m_\gamma \in \mathbb{IR}_d^\gamma$, there exists exactly one corresponding external representation m_d. A lack of bijectivity in the mapping between internal and external representations directly leads to a loss of information during the mapping.

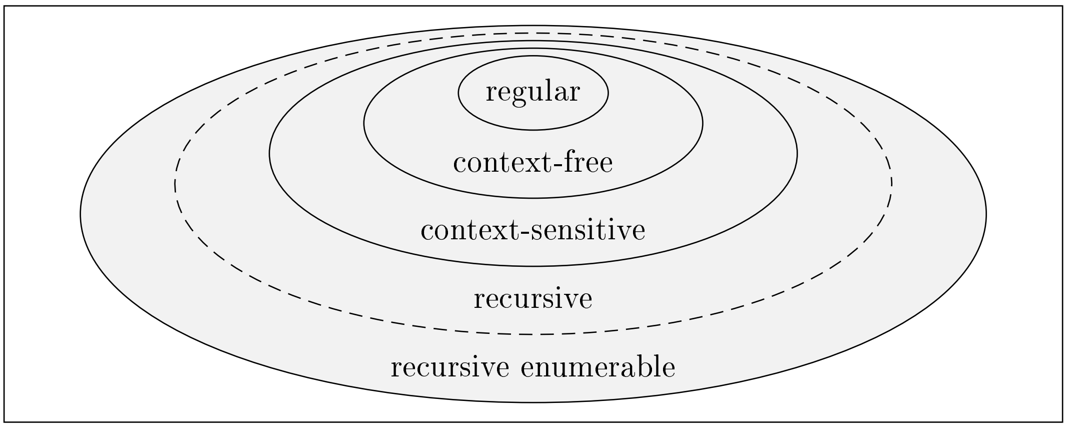


Figure 3.5: Containment of grammar classes, based on Chomsky [Cho59].

Requirement 3.5.5: *The mapping function $f_{\gamma\to d}$ defines a bijective mapping between the sets $\mathbb{ER}_d$ and $\mathbb{IR}_d^\gamma$ as one-to-one correspondence between both sets.*

For approaches on data format description, the question is whether or not it is possible to guarantee the satisfaction of aforementioned requirements. Satisfying Requirements 3.5.1 to 3.5.4 relates to issues of *computability*, *decidability* and *tractability* concerning the validation functions v_d^γ, v_d and concerning the mapping function $f_{\gamma\to d}$ including its inverse $f_{\gamma\to d}^{-1}$. Likewise, Requirements 3.5.2, 3.5.4 and 3.5.5 lead to the question of *one-to-one correspondence* between sets of external and internal representations $\mathbb{ER}_d$ and $\mathbb{IR}_d^\gamma$.

3.5.2 Computability and decidability of functions

Specifying a data format defines a normative set of external representations $\mathbb{ER}_d$, including the external validation function v_d. The set can be considered a formal language L, where in accordance to Requirement 3.5.3, there exists an automaton M that accepts or rejects every input, and terminates, thus *deciding* L. If an automaton M accepts L, without making any other statement on other inputs, it is *recognising* L.

Using established results from formal languages [HU79, Sip97, Cho59], it has to be shown which class of formal language can be used to model external representations of a data format, and whether it is possible to construct an automaton that decides membership in a language of that class:

- If L is a context-sensitive language, then there exists a *linear-bounded automaton (LBA)* M as minimal automaton which recognises and decides L.
- If L is a recursive language, then there exists a *Turing Machine (TM)* M as minimal automaton which recognises and decides L.
- If L is a recursively enumerable language, then there exists a TM M as minimal automaton which recognises L, but does not necessarily decide it.

- If L is a recursively enumerable, but not recursive language, then there exists no TM M as minimal automaton which both recognises and decides L.

Since there shall be an automaton M which decides the language L, L has to be at most recursive to satisfy Requirement 3.5.3, so an automaton M can exist which terminates. Yet, not every automaton terminates, which is required for deciding a language:

- If M is a TM, deciding whether M accepts a given input and terminates is the so-called *Accepting Problem* A_{TM} (Eq. 3.12), also known to be the undecidable *Halting Problem.* Undecidability of A_{TM} also precludes the existance of a class of automata which recognise and decide exactly the set of all recursive languages. If M is a TM, it can thus recognise and maybe decide recursively enumerable languages.
$$A_{TM} = \{\langle M, w\rangle \,|\, M \text{ is a TM and accepts } w\} \tag{3.12}$$

- If M is a LBA, then deciding whether M accepts a given input and terminates is the Accepting Problem A_{LBA} (Eq. 3.13) known to be decidable [Sip97]. In this case, M can recognise and decide at most context-sensitive languages.
$$A_{LBA} = \{\langle M, w\rangle \,|\, M \text{ is a LBA and accepts } w\} \tag{3.13}$$

So for deciding whether a given automaton M actually accepts a given input and terminates, L has to be at most context-sensitive. Since context-sensitive languages are a subset of the recursive languages as shown in the Chomsky containment hierarchy of formal language classes depicted in Figure 3.5, the use of context-sensitive languages would guarantee the existance of a terminating automaton M.

The remaining question to be answered is which class of language has to be supported for modelling the set of external representations for arbitrary data formats:

- Assuming a data format defining a context-sensitive language L for its external representations, there exists an LBA M that decides L, and it is possible to decide whether M terminates for a given input.

- Assuming a data format defining a recursive, but not context-sensitive language L for its external representations, there exists a TM M that recognises L, but not necessarily decides it. Moreover, it is not possible to decide whether M accepts a given input and terminates.

This question can thus be answered by successfully constructing at least one data format that defines a recursive but not context-sensitive language L for its external representations $\mathbb{ER}_d$.

THEOREM 3.5.1: *There exists at least one data format d which specifies a set of external representations $\mathbb{ER}_d$ which corresponds to a recursive, but not context-sensitive language.*

Proof. Assume a data format which serves for transmitting a long bit sequence in a message between sender and receiver. Whether or not a long bit sequence is valid in a message is decided by a separate function z shared by sender and receiver. The data format defines a set of external representations, where a valid bit sequence is compressed using an *Exponential Golomb* run-length compression scheme. In this scheme, a so-called *run* of 2^n consecutive bits all set to either 0 or 1 is represented through the *run length* n encoded using the Exponential Golomb integer encoding with a bit length of $2 \times log_2(n)$, consisting of two-bit pairs where the first bit states whether another pair follows, and where the second bit is part of the bit sequence which encodes n. Such a run length is followed by the bit value of the run, either 0 or 1.

The set of external representations forms a language L, for which to decide membership, an automaton M has to decompress the bit sequence prior to deciding its validity through z. Assuming this to be a context-sensitive language, there exists a LBA as automaton M which is also capable of decompressing the Exponential Golomb run-length encoding. As a LBA has only a limited amount of tape for processing, M has a linear upper bound $o \times p$ for processing compressed inputs of length p. For any o given, there exists a compressed bit sequence with length $p = 2 \times log_2(n)$ consisting of a single run with a decompressed length $2^n > o \times p$ to exceed its linear upper bound, so an LBA cannot decompress it and apply z for validation. A LBA is not capable of computing a decompression transformation where the length of the output is not strictly bounded by a linear function of the input length. $\square$

In this light, it might be tempting to require the definition of an upper bound n for a given data format d as the maximum length of bit sequences for its external representations $\mathbb{ER}_d$ to enable the use of LBAs, seemingly sufficient for data formats in practice. Yet, existing data formats used for *multimedia streaming* are good examples which do not have such an upper bound n, such as the MPEG-2 Transport Stream [ISO00] data format used for Digital Video Broadcasting over satellite (DVB-S), cable (DVB-C) or terrestrial radio (DVB-T).

Relating this theoretical result to practice, an Exponential Golomb run-length compression scheme, similar to the one used in the previous Proof 3.5.2, has been proposed for the compression of scan test data in system-on-a-chip (SOC) designs in literature [LC04], effectively to be used in a data format. As a consequence, one can either:

- choose LBA as type of automaton, where it can be decided whether an LBA terminates as required, but which fails to model languages that represent external representations of valid data formats, or

- choose TM as type of automaton, which can model all languages that represent external representations of valid data formats, but where it cannot be decided whether the TM terminates as required.

In general, describing arbitrary data formats requires a computational device with sufficient computational power to include the set of recursive languages. Due to the Halting Problem, such a device necessarily includes the set of recursively enumerable languages and thus is too powerful to guarantee decidability.

Corollary 3.5.1: *Considering data formats in general, decidability of external validation functions v_d cannot be guaranteed.*

In practice, computational devices have limited resources such as memory at their disposal, so these are effectively closer to being a LBA rather than a TM. For a data format d which defines a set of external representations $\mathbb{ER}_d$ that corresponds to a recursive, but not context-sensitive language, any implementation in practice can only handle a subset of format-compliant external representations which fits within its restricted computational resources.

3.5.3 Tractability of functions

Assuming validation and mapping functions to be computable and decidable, it remains to be seen whether the tractability of a mapping function and its inverse are related, or whether these functions are necessarily tractable at all. A problem is *tractable* if it is efficiently solvable in polynomial time by a deterministic TM, and *intractable* otherwise.

THEOREM 3.5.2: *For a data format d, neither are mapping functions and their inverses necessarily tractable, nor is their tractability related.*

Proof. Actual proof is given by a publication on inherently reversible grammars [Dym91]. The author states that every formal grammar G has six computational problems termed *p-acceptance*, *p-enumeration*, *g-acceptance*, *g-enumeration*, *bi-acceptance* and *bi-enumeration.* For a given grammar G and a word w, the p- and g-acceptance problems relate to deciding whether w can be parsed or generated, while p- and g-enumeration relates to enumerating the ways this can be done. The remaining bi-acceptance and bi-enumeration problems relate to deciding whether a word w conforms to a certain parse tree, and enumerating all corresponding pairs of words and parse trees.

Using this classification, the publication shows that finite p-enumeration does not entail finite g-enumeration and vice versa, by giving two examples based on Hilbert's tenth problem and the undecidability of first-order logic, where either g-acceptation or p-acceptation is not decidable. The results are extended towards tractability, where a tractable g-enumeration does not entail a tractable p-acceptation and vice versa. Examples given relate to number products in public key cryptography and NP-complete problems. $\square$

As a consequence, bijective mapping functions and their inverses are not necessarily tractable, nor is their tractability related.

3.5.4 One-to-one correspondence of sets

For a data format d, its set of external representations $\mathbb{ER}_d$ may be infinite. Due to Requirement 3.5.5 on the one-to-one correspondence of internal and external representations, a sender or receiver γ thus has its set of internal representations $\mathbb{IR}_d^\gamma$ which may be infinite as well.

The questions to be answered are whether one-to-one correspondence between $\mathbb{ER}_d$ and $\mathbb{IR}_d^\gamma$ can be tested, or whether it can be guaranteed by construction.

THEOREM 3.5.3: *Testing one-to-one correspondence between infinite sets* $\mathbb{ER}_d$ *and* $\mathbb{IR}_d^\gamma$ *as defined by a given function* $f_{\gamma \to d}$ *is undecidable.*

Proof. Assume a TM $M_{d\to\gamma}$ that decides whether $f^{-1}_{\gamma\to d}$ is injective, and a TM $M_{\gamma\to d}$ that decides whether $f_{\gamma\to d}$ is injective. If both TMs accept their respective mapping function, then there exists a one-to-one correspondence between $\mathbb{ER}_d$ and $\mathbb{IR}^\gamma_d$ which satisfies Requirement 3.5.5.

Further assume that for every element $m_d \in \mathbb{ER}_d$, the TM $M_{d\to\gamma}$ computes $m_\gamma = f^{-1}_{\gamma\to d}(m_d)$ and compares m_γ to all computed elements. If there exists $m'_\gamma = f^{-1}_{\gamma\to d}(m'_d)$ with $m_\gamma = m'_\gamma$, then the TM decides the problem by rejecting it. In that case, m_γ maps to both m_d and m'_d, which violates injectivity, and thus one-to-one correspondence.

In case of a finite set $\mathbb{ER}_d$, $M_{d\to\gamma}$ is decidable, as it eventually tests all elements of the set. In case of a countably infinite set, the TM is only semi-decidable, as it only terminates when rejecting the mapping function, yet loops otherwise.

The situation is analogous for its twin $M_{\gamma\to d}$, and for infinite sets $\mathbb{ER}_d$ and $\mathbb{IR}^\gamma_d$, both TMs only terminate in case of a non-bijective function $f_{\gamma\to d}$, otherwise looping forever. Thus, the problem of testing one-to-one correspondence between infinite sets is undecidable. □

COROLLARY 3.5.2: *Testing one-to-one correspondence as defined by a given mapping function $f_{\gamma\to d}$ between finite subsets of $\mathbb{IR}^\gamma_d$ and $\mathbb{ER}_d$ is decidable.*

As it is not possible to test arbitrary functions whether they provide for a bijective mapping between potentially infinite sets $\mathbb{ER}_d, \mathbb{IR}^\gamma_d$, the question remains whether bijectivity of a mapping function can be guaranteed by construction.

THEOREM 3.5.4: *Guaranteeing one-to-one correspondence between infinite sets $\mathbb{ER}_d$ and $\mathbb{IR}^\gamma_d$ by construction of a function $f_{\gamma\to d}$ is decidable.*

Proof. Assume a so-called Reversible Turing Machine (RTM) M for computing a bijective mapping between $\mathbb{ER}_d$ and $\mathbb{IR}^\gamma_d$. By definition, its transition function σ is required to be injective for determinism of M, and is required to be surjective for reversibility of M. As σ is composed from a finite set of transition formulas, injectivity and surjectivity of transition formulas and thus of σ can be decided. Informally speaking, bijectivity of transition formulas forces an RTM to retain all information during computations.

So one-to-one correspondence of sets can be guaranteed by constructing a bijective mapping function using an RTM, but its computational power may still be in doubt. Although reversibility of computational devices was initially believed to restrict their computational power [Lan61], it has been shown twice independently that a Reversible Turing Machine is Turing-complete and therefore capable of handling arbitrary computable bijective mapping functions [Lec63, Ben73]. □

These results can be translated back to the domain of data format description. By constructing encodings and mapping steps as RTMs and thus guaranteeing their bijectivity, it is possible to construct bijective mappings between sets of external and internal representations $\mathbb{ER}_d$ and $\mathbb{IR}^\gamma_d$ which are guaranteed to retain information. Yet, due to being Turing-complete, RTMs still cannot be decided to terminate.

Implementing such a bijective mapping in practice would lead to “uniformity of implementation” regarding parsing and generation [Dym91], and thus be a desirable property in itself. Actually developing such bijective mapping functions using high-level reversible programming languages is a non-trivial task which is not the subject of this thesis. Due to numerous contributions to the research domain of *Reversible Computing*, there exist reversible programming languages like *Janus* [YG07] and *R* [Fra97], as well as the reversible computer architecture *Pendulum* [Vie95].

3.5.5 Summary

Satisfying the Requirements 3.5.1 to 3.5.5 is only partially within the hands of an approach to data format description, due to limits rooted in formal languages:

- It is not possible to both have a sufficiently expressive model capable of expressing external representations from arbitrary data formats and still guarantee the termination of validation functions. Any sufficiently expressive approach to describe arbitrary data formats has to resort to heuristics in order to decide whether a validation function will terminate, possibly using limitations of the physical computational device itself, such as limited memory resources.

- Tractability of mapping functions and their inverses is not given, nor is tractability of a mapping function and its inverse necessarily related, as can be seen from the existence of "trapdoor" functions in cryptography. A practical example of such a "trapdoor" function is the computation of the product of two large prime numbers, which can be computed efficiently. Its inverse corresponds to the yet-as-unsolved problem of factorising large integer numbers efficiently, which serves as a basis for widespread cryptographic approaches such as RSA.

- Yet, one-to-one correspondence between internal and external representations can be guaranteed by construction of Turing-complete RTMs which are information-preserving, leading to future research regarding the application of reversible programming languages in implementing encodings and transformations for data format descriptions.

3.6 Discussion

The formalization presented in this chapter provides a foundation for defining a new approach on describing data formats that directly uses the identified elemental descriptive capabilities of decoding, segmentation, transforming and concatenation. Still notable is the *need for a canonical internal representation* in order to define a data format.

- **Need for a canonical internal representation:** As can be learned from the formalization, the same external representation m_d may be represented internally as m_γ in substantially different ways, depending on the sender or receiver γ. Since one-to-one correspondence can be provided through RTMs, the number of corresponding internal representations is only limited by the number of total RTMs, leading to the need of a canonical internal representation. The constraint of bijectivity between an external representation and its corresponding internal representation does not reduce the number of potential internal representations for a given external representation.

Rather than covering the complexity of mapping between arbitrary internal and external representations with an infinite amount of variations for a given data format, it is more feasible to introduce a *canonical internal representation.* This is without loss of generality, as arbitrary internal representations can still be obtained through a separate stage of bijective mapping, for example for representing the same information at a different granularity, yet it enables the definition of a model for such a canonical internal representation.

3.7 Summary

This analysis has introduced the research hypothesis 3.2.1, which states that the current State of the Art regarding data format description can be improved by assuming a data format to define a normative set of lossless information representations. According to the hypothesis, the mapping between internal and external representations is information-preserving, thus bijective and therefore lossless. Building on the research hypothesis, the resulting formalisation leads to the notion of a causality graph for describing the composition of data.

Four important properties of the formalisation were explored, of which the first three are the suitability of bijective mapping functions for data formats, the sufficiency for lossless and lossy data formats, as well as the sufficiency and necessity of its descriptive capabilities for handling primitive, structured, transcoded and fragmented data. This establishes the four elementary descriptive capabilities of decoding primitive data, segmenting structured data, transforming transcoded data and concatenating fragmented data, which exactly match the descriptive capabilities that were considered during the survey of current State of the Art. Last but not least, the remaining property that was explored is a “litmus test” for data format description approaches, which is a specific PNG raster image that requires the support of all four elemental descriptive capabilities from a data format description approach.

Using the formalisation to explore the limits to data format description, all approaches are limited by established theoretical restrictions from formal languages and computational theory to provide support for arbitrary data formats, yet guarantee the satisfaction of basic requirements.

From the comparison of approaches from related work in Chapter 2, effectively based on the elementary descriptive capabilities, only XCEL comes close to being universally applicable, yet it still lacks support for bit granularity and full support for the concatenation of fragmented data with full control over fragment ordering. The majority of other examined approaches fails to support transcoded or fragmented data at all.

The formalisation presented in this analysis contributes a solid foundation for universal applicability, and thereby paves the way for a model for formally describing data format instances in the upcoming Chapter 4, which addresses the need for a canonical internal representation that was discussed previously. In turn, this upcoming model enables a model for formally describing data formats in a declarative manner in Chapter 5, which is formally robust and conceptually simpler than XCEL.

Chapter 4

Describing Data Format Instances

4.1 Introduction

In the previous Chapter 3, a formalised abstraction of data format instances and data formats was presented and subjected to an analysis regarding its inherent properties, which affects data format description and its applications. The presented abstraction of data format instances consists of a causality graph rooted in the original bitstream and where contained information corresponds to primitive values as its leaves. The abstraction is a suitable basis for a model of describing data format instances to be used in manual and automated settings.

This chapter presents the *Bitstream Segment Graph (BSG)* model and addresses the following aspects:

- **Definition of the BSG model:** Besides the definition of the BSG model in Section 4.2, this also addresses means of incremental construction and modification of BSG instances.

- **Representation of BSG instances:** Both a visual representation and a storage representation based on the *Resource Description Framework (RDF)* are given in Section 4.3.

- **Construction and modification of BSG instances:** For incremental construction and modification of BSG instances, a closed set of operations is presented in Section 4.4. Based on these operations, the Apeiron BSG Editor offers tool support for constructing, modifying and exploring BSG instances on arbitrary data.

- **Applications of the BSG model:** Section 4.5 demonstrates the description of the PNG image "litmus test" from Chapter 3, which exercises all elemental descriptive capabilities. Furthermore, the section gives an application of BSG models for describing exploits from IT Security, which is presented in detail.

4.2 Definition of the Bitstream Segment Graph model

As per Definition 3.3.15, a data format instance is a causality graph composed from transformations, with encodings defined for its leaf nodes. In combination with Definition 3.3.14, arbitrary causality graphs can be composed from the mapping steps of segmentation, transformation and concatenation.

4.2.1 Defining codings and transformations

Referring to coding and transforming as elemental descriptive capability for which the actual mapping has to be defined in "executable" terms, the following definitions are introduced:

Definition 4.2.1 (Bitstream Coding Function): A *bitstream Coding function* is a bijective mapping function between a domain-specific value and its corresponding bit sequence. It can be uniquely identified and may have additional parameters.

Definition 4.2.2 (Bitstream Coding): A *bitstream Coding* represents a bijective mapping between a domain-specific value and its bit sequence in a bitstream segment, using a bitstream coding function. It identifies the used bitstream coding function and its parameters, if any.

Example 4.2.1: Let $x \in \mathbb{N}$ be the width of an image in pixels. If x is to be represented as bit sequence in a *least-significant bit first unsigned integer* encoding, then a corresponding bitstream coding function f is required.

A bit sequence can represent an encoded typed value that is part of the information stored in a message such as a file or a protocol data unit (PDU), a message exchanged in a network protocol. For example, there are two bit sequences contained within a PNG raster image file which contain encoded integer values that represent the width and height of the image.

Definition 4.2.3 (Bitstream Transformation Function): A *bitstream transformation function* is a bijective mapping function between two bit sequences as input and output. It can be uniquely identified and may have additional parameters.

Definition 4.2.4 (Bitstream Transformation): A *bitstream transformation* represents a bijective mapping between two bit sequences as input and output, as produced by a bitstream transformation function. It identifies the used bitstream transformation function and its parameters, if any.

Example 4.2.2: Let a finite bit sequence a represent a data structure that is encrypted using the RC4 stream cipher with a secret key k. Prior to segmenting the structure, decrypting the bit sequence is required, which is a bitstream block transformation. It requires a corresponding function which implements the RC4 stream cipher and uses the secret key k as parameter.

Example 4.2.3: The bit-wise inversion of a finite bit sequence requires a bitstream transformation function as block transformation and uses no parameters.

Both functions for bitstream coding and bitstream transformation can be defined through Reversible Turing Machines (RTM) and referred to by a unique identifier per RTM. It is not within the scope of the BSG model to execute arbitrary bitstream codings and transformation functions itself.

4.2.2 Defining bitstream segments

Definition 4.2.5 (Bitstream Segment): A *bitstream segment* represents a finite bit sequence.

Definition 4.2.6 (Bitstream Source): A *bitstream source* is a bitstream segment whose bit sequence represents a digital item which is composed according to a data format, and which is to be described.

Example 4.2.4: Examples for octet-aligned bitstream sources are files, network packets or file systems on a storage medium.

Both codings and mapping steps define the structural meaning of bitstream segments as nodes in the causality graph. They give rise to *structural types* of bitstream segments.

Definition 4.2.7 (Bitstream Segment Type): A *bitstream segment type* defines the structural purpose of a bitstream segment. Every bitstream segment belongs to exactly one of 6 bitstream segment types, which is either a *structure*, a *transcode*, a *fragment*, a *composite*, a *primitive* or a *generic*:

- A *structure* bitstream segment is input to a segmentation, which separates the bit sequence into two or more elements.
- A *transcode* bitstream segment is input to a bitstream transformation function, which transcodes the bit sequence into another one.
- A *fragment* bitstream segment is input to a concatenation, which concatenates the bit sequences with that of other fragments.
- A *composite* bitstream segment is output of a concatenation, and is the concatenation of bit sequences from two or more fragment bitstream segments.
- A *primitive* bitstream segment is input to a bitstream coding function, which results in a typed primitive value.
- A *generic* bitstream segment is neither input to a mapping step nor to an encoding. Extending the set of types previously shown in Chapter 3.4.3, this type is required for incremental description of a data format instance, and in case of incomplete data format knowledge, it acts as a temporary placeholder.

4.2.3 Defining a Bitstream Segment Graph

Maintaining exactly one bitstream segment type for every bitstream segment would fail for a bitstream segment, which is simultaneously a composite and another "downward" type, such as when concatenated fragments of data are further transformed. This issue is resolved through maintaining *normalisation* of BSG instances.

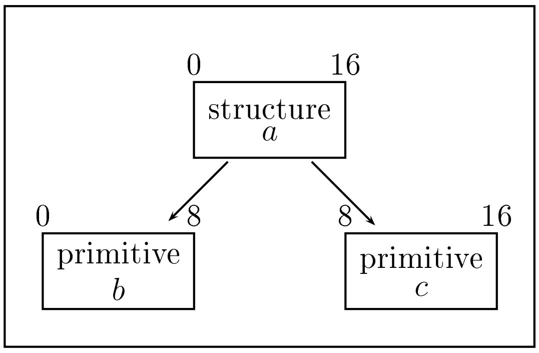


Figure 4.1: Representation of bitstream segments in the *simple* variant, showing a structure bitstream segment a containing two primitive bitstream segments b and c.

Definition 4.2.8 (Normalisation): In a *normalised* causality graph, every composite bitstream segment has a single successor bitstream segment which is assignated the "downward" type (other than composite) as required to designate its structural type.

Definition 4.2.9 (Bitstream Segment Graph): A *bitstream segment graph (BSG)* is a normalised causality graph composed from bitstream segments.

4.3 Representation of BSG instances

For BSG instances, some means for their representation is required, both for representing them *visually* for use in print and for representing them *digitally* for use in applications such as editors.

4.3.1 Visual representations

Depending on the requirements of visual representations of whole BSG instances as well as bitstream segments, one of three variants for visual representations (*simple*, *extended* and *interactive*) can be used. While simple and extended visual representations are for use in print, the interactive visual representation is for use in applications.

Simple visual representation

In this variant, segments are depicted as boxes, and the relations of 'predecessorship' and 'successorship' are depicted by directed arrows from predecessors to successors. Actual placement of segments in the bitstream segment is given relative to a predecessor, or zero-based for a root bitstream segment. For a segment, inclusive start and exclusive end bit positions are shown above the upper left and right corners, respectively. The bitstream segment type is given in the upper half of the box, while an identifier for the actual segment is located in the lower half. An example of this variant for visual representation is shown in Figure 4.1.

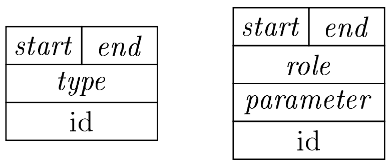


Figure 4.2: Representation of bitstream segments in the *extended* variant, showing templates for generic, structure and composite bitstream segments (left) and for fragment, primitive and transcode bitstream segments (right)

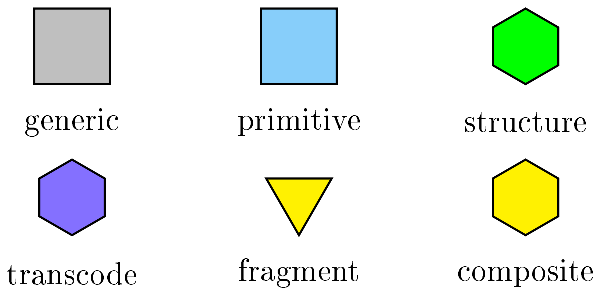


Figure 4.3: Representation of bitstream segments in the *interactive* variant, showing symbols for all bitstream segment types.

Extended visual representation

As an alternative to the simple variant, the extended visual representation provides a more compact notation for showing BSG instances in print.

Segments are depicted as three-row boxes for generic, structure and composite bitstream segments, and as four-row boxes for fragment, primitive and transcode bitstream segments, as shown in Figure 4.2. The additional row in the latter case serves to provide additional information, such as an ordering index for fragments, or references to bitstream coding and transformation functions. As with the simple variant, placement of bitstream segment is relative to predecessors, with the inclusive start position and exclusive end position shown in the upper left and right corners, respectively.

Interactive visual representation

For showing a BSG instance in an interactive application, where properties of a bitstream segment do not have to be shown all at once, but can be accessed in some other means, a less complex visualisation can be used. In this representation variant, bitstream segments are visualised as coloured shapes shown in Table 4.3.

Prefix	Description / namespace
`rdf`	RDF namespace: `http://www.w3.org/1999/02/22-rdf-syntax-ns#`
`bsg`	Namespace for BSG-related RDF vocabulary: `http://dataformats.net/bsg/1.0/`
`bsge`	Example prefix for bitstream coding function identifiers
`bsgt`	Example prefix for bitstream transformation function identifiers
`png`	Example prefix for PNG format-specific semantics identifiers

Table 4.1: Namespace declarations.

4.3.2 Digital representation

In order to exchange information on the composition of binary data, a Bitstream Segment Graph instance can be expressed through RDF/N3 using the RDF defined in this subsection. In the following definitions and examples, RDF namespaces and their prefixes are defined according to Table B. Using the BSG RDF vocabulary, bitstream segments are represented as resources which belong to certain *RDF classes* and have certain *RDF properties*:

- **RDF Classes:** Bitstream segments are distinguished in their type through RDF classes. Every bitstream segment has a `rdf:type` of both `bsg:segment` and the specific RDF class corresponding to its type as listed from Table 4.3, such as `bsg:primitive`. A root bitstream segment additionally has a `rdf:type` of `bsg:source`. It is worth noting that the normalised bitstream transformations of segmentation, block transformation and concatenation from Definition 3.3.14 correspond to the classes `bsg:structure`, `bsg:transcode` and `bsg:composite`, respectively.

- **RDF Properties:** Depending on the RDF class, bitstream segments have specific properties according to Table 4.4. For placement, every bitstream segment has a `bsg:start`, `bsg:length` and `bsg:end` property with integer values. These refer to its exact placement within the bit sequence composed from its predecessor(s), or within its defined bit sequence in case of a bitstream source. A root bitstream segment always starts at 0. All three properties are measured in bits, whereas the start position is included and the end position excluded.

  Regarding their composition, every bitstream segment besides the bitstream source has an ordered list of one or more `bsg:predecessor` properties, and every bitstream segment besides `bsg:generic` or `bsg:primitive` segments has an ordered list of one or more `bsg:successor` properties. Class-specific restrictions listed in Table 4.4 apply which correspond to the underlying BSG model. Only a bitstream source may have the `bsg:source` property set.

  The meaning of a bitstream segment can be assigned through zero or more `bsg:semantics` properties. For example, this could refer to *PNG Signature* semantics using `png:signature` as value. For `bsg:primitive` and `bsg:transcode`

Used in coding?	Used in transformation?		Type	rdf:type
no	no	(as input)	Generic	`bsg:generic`
yes	no	(as input)	Primitive	`bsg:primitive`
no	segmentation	(as input)	Structure	`bsg:structure`
no	transformation	(as input)	Transcode	`bsg:transcode`
no	concatenation	(as input)	Fragment	`bsg:fragment`
no	concatenation	(as output)	Composite	`bsg:composite`

Table 4.2: Bitstream segment types.

rdf:class	Description
`bsg:source`	Class for bitstream sources
`bsg:segment`	Abstract base class for bitstream segments
`bsg:generic`	Class for bitstream segments where the purpose is undefined
`bsg:primitive`	Class for bitstream segments representing an encoded literal
`bsg:structure`	Class for bitstream segments composed from two or more bitstream segments with separate, distinct meanings
`bsg:transcode`	Class for bitstream segments representing a transcoded bit sequence
`bsg:fragment`	Class for bitstream segments representing a fragment of a larger bit sequence with a uniform meaning
`bsg:composite`	Class for bitstream segments representing a bit sequence with a uniform meaning aggregated from two or more fragments

Table 4.3: RDF classes for bitstream segments.

bitstream segments, the identification of the actual bitstream coding or transformation function used is given through the `bsg:encoding` and `bsg:codec` properties, respectively. For example, this could include a *most significant bit first unsigned integer* encoding `bsge:msbf-uint` or a *gzip* transformation `bsgt:gzip`. The definition of concrete identifiers for semantics, encodings and codecs strongly depends on the data format to be described and thus is out of scope for this thesis.

4.4 Construction and modification of BSG instances

A trivial BSG instance can be created by defining a single generic bitstream segment, which is a bitstream source representing the bit sequence to be described. Through further modification of the BSG instance, the description can be improved step-by-step.

Class	Property	Cardinality	Description
`bsg:source`	`bsg:href`	1..1	Reference to a bitstream source
`bsg:segment`	`bsg:start`	1..1	Start position in bits (inclusive)
	`bsg:length`	1..1	Length in bits
	`bsg:end`	1..1	End position in bits (exclusive)
	`bsg:semantics`	0..n	Identifier for format-specific semantics
	`bsg:predecessor`	0..n	Ordered list of predecessors (input)
	`bsg:successor`	0..n	Ordered list of successors (output)
`bsg:generic`	`bsg:predecessor`	*0..1*	*Restriction: Generics have at most one predecessor*
	`bsg:successor`	*0..0*	*Restriction: Generics do not have successors*
`bsg:primitive`	`bsg:encoding`	1..1	Identifier for the bitstream coding function used
	`bsg:predecessor`	*0..1*	*Restriction: Primitives have at most one predecessor*
	`bsg:successor`	*0..0*	*Restriction: Primitives do not have successors*
`bsg:structure`	`bsg:predecessor`	*0..1*	*Restriction: Structures have at most one predecessor*
	`bsg:successor`	*2..n*	*Restriction: Structures have at least two successors*
`bsg:transcode`	`bsg:codec`	1..1	Identifier for the bitstream transformation function used
	`bsg:predecessor`	*0..1*	*Restriction: Transcodes have at most one predecessor*
	`bsg:successor`	*1..1*	*Restriction: Transcodes have exactly one successor*
`bsg:fragment`	`bsg:predecessor`	*1..1*	*Restriction: Fragments have exactly one predecessor*
	`bsg:successor`	*1..1*	*Restriction: Fragments have exactly one successor*
`bsg:composite`	`bsg:predecessor`	*2..n*	*Restriction: Composites have at least two predecessors*
	`bsg:successor`	*1..1*	*Restriction: Composites have exactly one successor*

Table 4.4: RDF properties for bitstream segments.

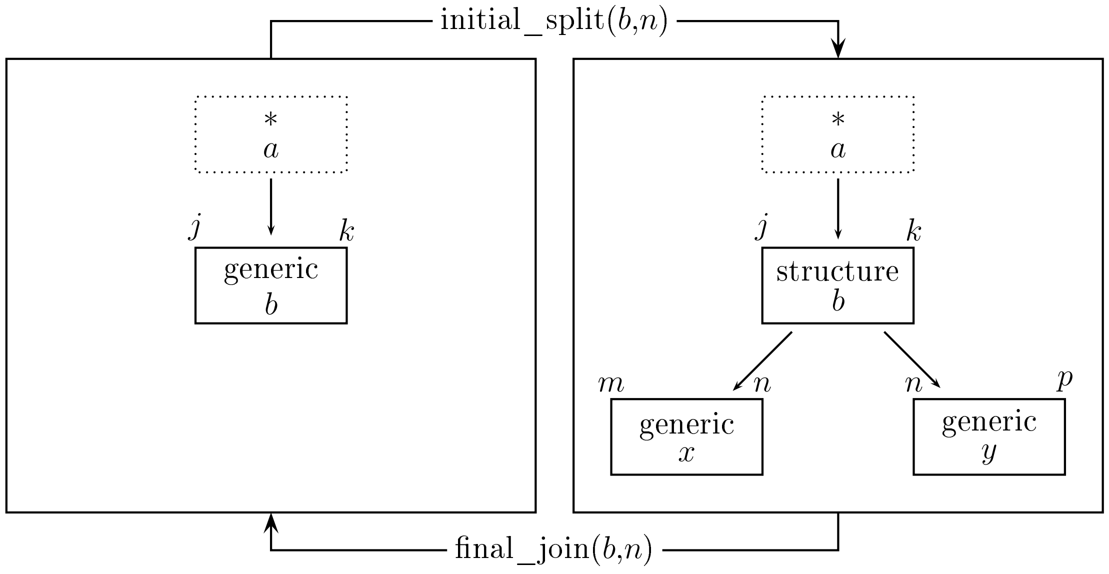


Figure 4.4: Graph grammar rule for initial_split/final_join operations

4.4.1 Modifying BSG instances through operations

To modify a BSG instance, for example when constructing a BSG instance incrementally, a notion of *operation* is required.

DEFINITION 4.4.1 (OPERATION): An *operation* o transforms a valid BSG instance into another valid BSG instance. For every operation o, there exists an inverse operation o^{-1} which undoes the effect of o.

This thesis defines a number of operations listed with their inverses for incremental construction of BSG instances, namely *initial_split* / *final_join*, *split* / *join*, *tie* / *untie*, *declare_primitive* / *undeclare_primitive*, *declare_fragment* / *undeclare_fragment*, *compose* / *decompose* and *expand* / *compress*.

In order to visualise these operations, they are shown as graph grammar rules using the simple visual representation previously defined in Section 4.3.1. In addition to their textual descriptions below, these operations are visualised accordingly in Figures 4.4 to 4.10, and detailed with examples using the PNG raster image file format test case which served as "litmus test" in Chapter 3:

- **initial_split, final_join**: Given a generic bitstream segment x, the operation creates two separate, neighbouring generic bitstream segments y, z by splitting the bit sequence of x in two at bit position p. It then changes x to a structure bitstream segment and adds y, z as successors of x, as shown in Figure 4.4.

  EXAMPLE 4.4.1: Let x be a generic bitstream segment which contains a valid PNG image. Applying the initial_split operation on x at bit position 64 results in a structure bitstream segment x with two successors, a bitstream segment y containing the PNG signature and a bitstream segment z containing three or more PNG chunks for a valid PNG image.

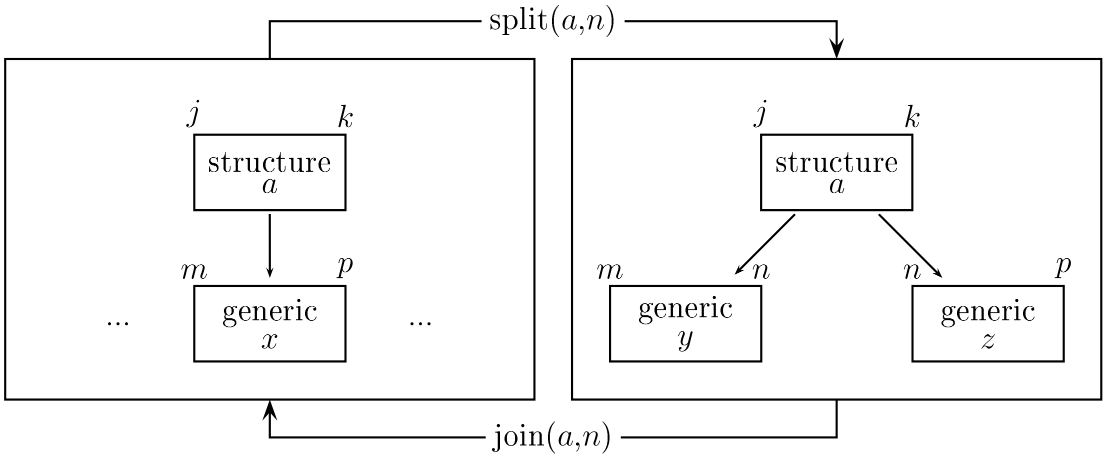


Figure 4.5: Graph grammar rule for split/join operations

- **split, join**: Given a generic bitstream segment x as a successor of a structure bitstream segment y, the operation splits x at bit position p into two separate generic bitstream segments x_1, x_2, and replaces x with x_1, x_2 as successors of y in-place, as shown in Figure 4.5.

  Example 4.4.2: Let x be the structure bitstream segment from Example 4.4.1, partially describing a valid PNG image, and let z be the second successor of x. Applying the split operation on z at bit position 200 results in a structure bitstream segment x with three successors, namely a generic bitstream segment y containing the PNG signature, a generic bitstream segment z_1 containing the IHDR chunk data structure, and a generic bitstream segment z_2 containing two or more PNG chunks. The bitstream segments z_1 and z_2 are on the same level as y, as opposed to the result of an initial_split operation.

- **tie, untie**: Given a structure bitstream segment x and a continuous subset of its successors as bitstream segments $a_0, \ldots, a_n$ as neighbours, the operation replaces the successors $a_0, \ldots, a_n$ with a structure bitstream segment b and adds $a_0, \ldots, a_n$ as only successors of b, as shown in Figure 4.6.

  Example 4.4.3: Let a_0, a_1, a_2 and a_3 be separate generic bitstream segments that have been the result of segmenting parts of a PNG image, but which have previously not been considered to form an *IHDR chunk* data structure. Through the tie operation, a_0 to a_3 remain generic bitstream segments, but are grouped under a structure bitstream segment b as successors that represent the IHDR chunk.

- **declare_primitive, undeclare_primitive**: Given a generic bitstream segment x and a reference to a bitstream coding function f, the operation replaces x with a primitive bitstream segment y, as shown in Figure 4.7.

  Example 4.4.4: Let x be the generic bitstream segment contained in the IHDR data structure from the previous Example 4.4.3, which contains the width of the

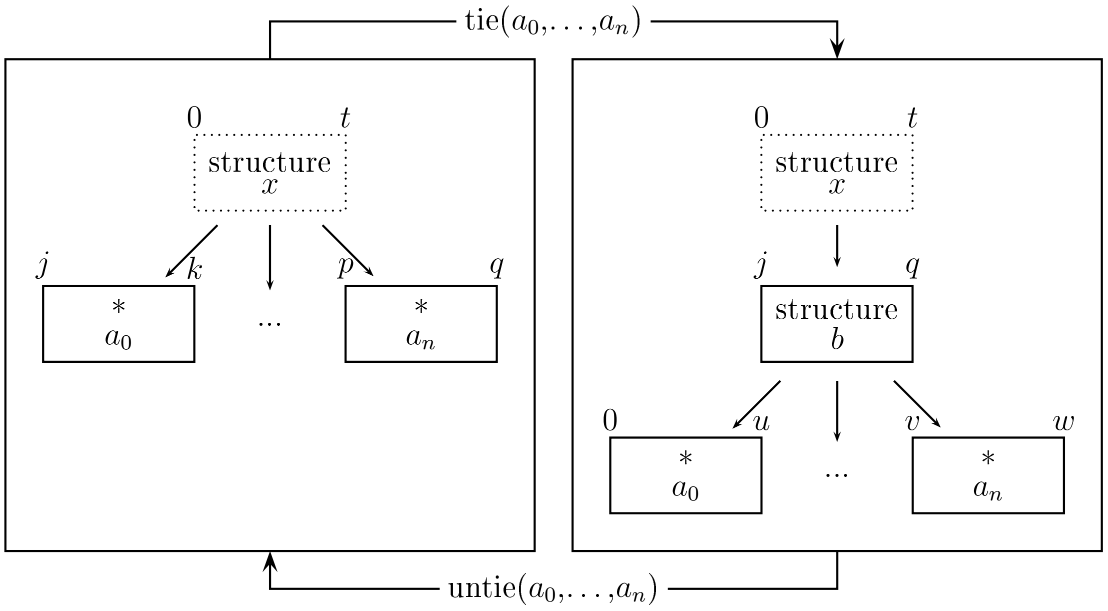


Figure 4.6: Graph grammar rule for tie/untie operations

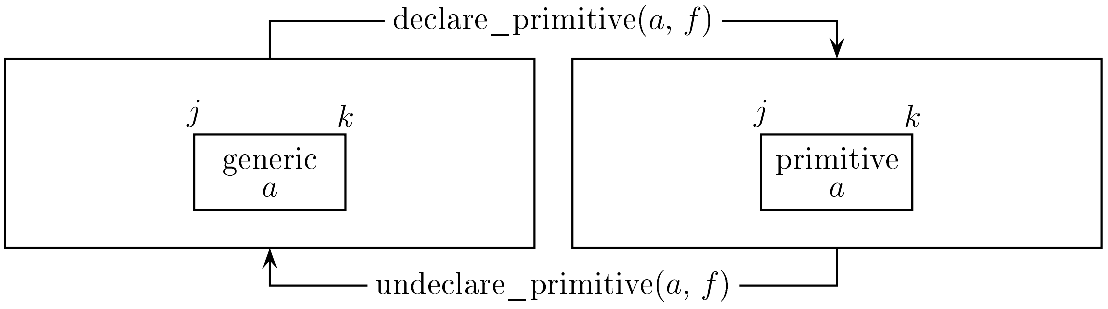


Figure 4.7: Graph grammar rule for declare_primitive/undeclare_primitive operations

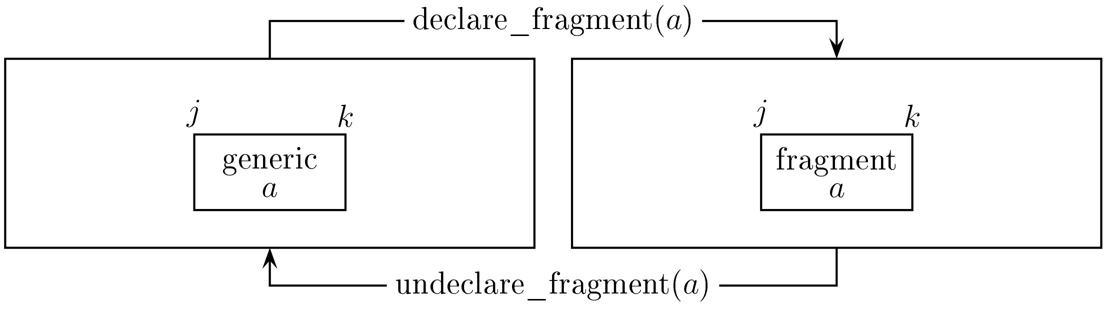


Figure 4.8: Graph grammar rule for declare_fragment/undeclare_fragment operations

PNG raster image in pixels. Through the declare_primitive operation on x and refering to a bitstream coding function for the *most-significant-bit-first integer encoding*, the bit sequence of x becomes the representation of an integer.

- **declare_fragment, undeclare_fragment**: Given a generic bitstream segment x, the operation replaces x with a fragment bitstream segment y, as shown in Figure 4.8.

  Example 4.4.5: Let a_0 and a_1 be two generic bitstream segments that represent the payloads of two separate IDAT chunks, which have been segmented previously. Both payloads represent fragmented image data, and as the fragments are not located next to each other, the segments need to be concatenated for further analysis. Applying the declare_fragment operation on both segments converts them to fragment bitstream segments that are ready for later concatenation.

- **compose, decompose**: Given an ordered set of fragment bitstream segments $\mathbb{X}$, the operation concatenates all bitstream segments in $\mathbb{X}$ in the given order to form a composite bitstream segment y, adding y as successor to all bitstream segments in $\mathbb{X}$. Furthermore, a generic bitstream segment z is added as sole successor to y, which allows further operations to be executed on z. The operation itself is shown in Figure 4.9.

  Example 4.4.6: Let a_0 and a_1 be the fragment bitstream segments resulting from the previous example 4.4.5. Applying the compose operation to the ordered set (a_0, a_1) produces the composite bitstream segment b, followed by a single generic bitstream segment x. This generic bitstream segment carries PNG image data that is still compressed and transformed, and thus subject to further operations.

- **expand, compress**: Given a generic bitstream segment x and a reference to a bitstream transformation function f, the operation replaces x with a transcode bitstream segment y and adds a generic bitstream segment z as its sole successor with a transformed bit sequence, as shown in Figure 4.10. The generic bitstream segment z contains the transcoded bit sequence of y.

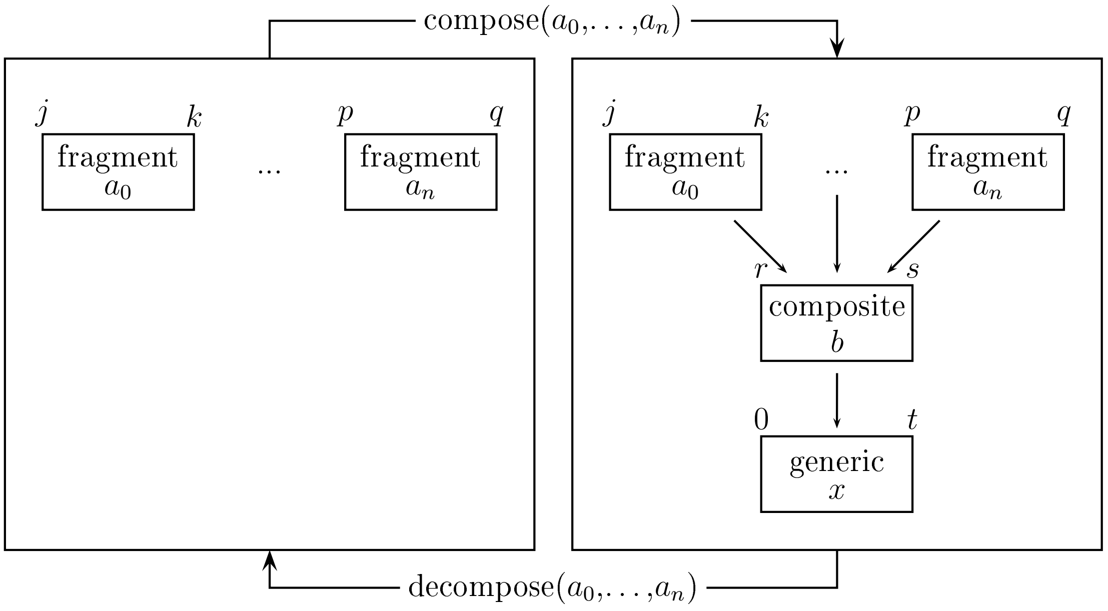


Figure 4.9: Graph grammar rule for compose/decompose operations

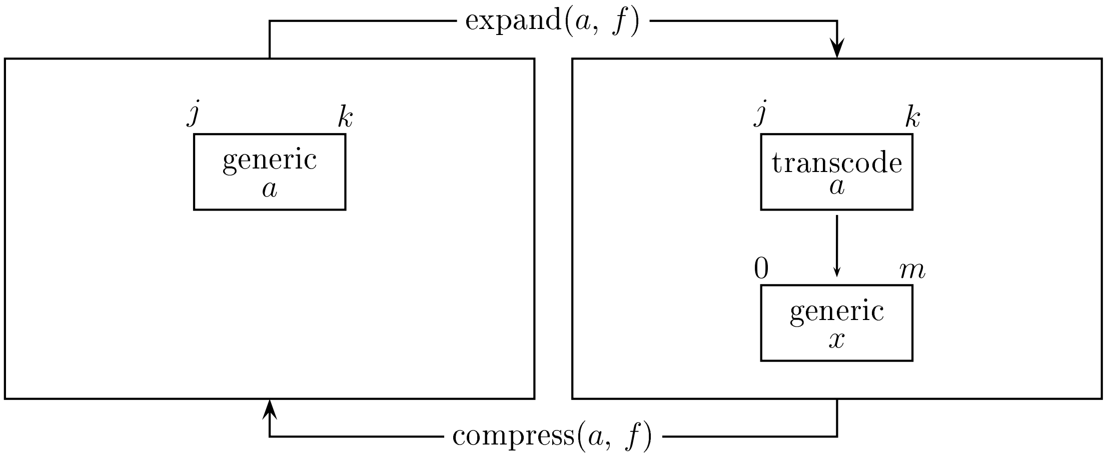


Figure 4.10: Graph grammar rule for expand/compress operations

Example 4.4.7: Let a be the generic bitstream segment resulting from the compose operation in the previous Example 4.4.6. Applying an expand operation using a reference to the bitstream transformation function for the *GZIP block transformation* turns a into a transcode bitstream segment, with a generic bitstream segment x as its sole successor that contains data that still has to be scanline-transformed in order to access actual pixels of the raster image.

Since the expand / compress operations are the only ones where the mapping between input and output bitstream segments may result in an increase or decrease of their actual total length, the names for both operations have been chosen deliberately in relation to compression, where changes in total length are actually desired. The use of the expand / compress terminology allows to uniquely identify both directions of mapping, yet a bitstream transformation function f used in expand / compress operations is neither required to be a compression-related transformation, nor to change the length of the bitstream segment at all. An example bitstream transformation function that does not change the length of bitstream segments would be a function performing bit-wise negation.

Using operations, a BSG instance can be constructed in a *divide and conquer strategy.* While they provide means for all four elemental descriptive capabilities, other combined operations could be envisioned that may further simplify manipulation of BSG instances. An example would be a *pushdown / pullup* operation, which would allow to "push" a bitstream segment a, which either leads or follows a structure bitstream segment b, "down" into the structure b as its first or last successor, respectively, as well as the inverse operation.

Example 4.4.8: Let a_0, a_1, a_2 and a_3 be bitstream segments which represent the length, the type, the payload and the CRC of a PNG chunk in that order. Let b be a structure bitstream segment which ties a_0, a_1 and a_2 as its successors, and let b be followed by a_3 as the result of a user error during annotation, who forgot to tie a_3 as well. In order to turn the structure bitstream segment b into a valid PNG chunk structure, some change is necessary. Rather than reversing the previous tie operation and repeating it correctly on a_0, a_1, a_2 and a_3, "pushing down" a_3 into b leads to the same result and corrects the erroneous annotation.

While a pushdown / pullup operation would simplify that specific case, it is possible to handle that case with the existing operations by untying b and again tying all previous successors of b plus / minus a again.

4.4.2 Measuring completeness of a description

Constructing a complete BSG instance from a generic segment representing the external representation to be described brings along the issue of *measuring the completeness* of its description.

Definition 4.4.2 (Coverage): The *coverage* of a bitstream segment is a measure between 0.0 and 1.0 which refers to the degree in which the segment and its successors describe its bit sequence through primitive values. A generic bitstream segment has a coverage of 0.0, while a primitive bitstream segment has a coverage of 1.0. The coverage of other types of bitstream segments is computed from the coverage of its successors and the percentage of their size with respect to all successors. The coverage of a BSG instance refers to the coverage of its root node.

EXAMPLE 4.4.9: Let x be a structure bitstream segment with a length of 16 bits, which contains two bitstream segments y, z as its successors. Both y and z have a length of 8 bits, where y is a primitive with coverage 1.0 and z is a generic with coverage 0.0. As a consequence, the coverage of x is $8/16 * 1.0 + 8/16 * 0.0 = 0.5$.

Depending on the coverage of its root bitstream segment, a BSG instance is either *partial* or *complete*:

- A BSG instance with a coverage of less than 1.0 is *partial*, as it still contains one or more generic bitstream segments, for which no more specific structural type has been assigned yet.
- A BSG instance with a coverage of 1.0 is *complete*, as it completely describes its root bitstream segment in terms of primitive bitstream segments through a finite number of bijective mapping steps.

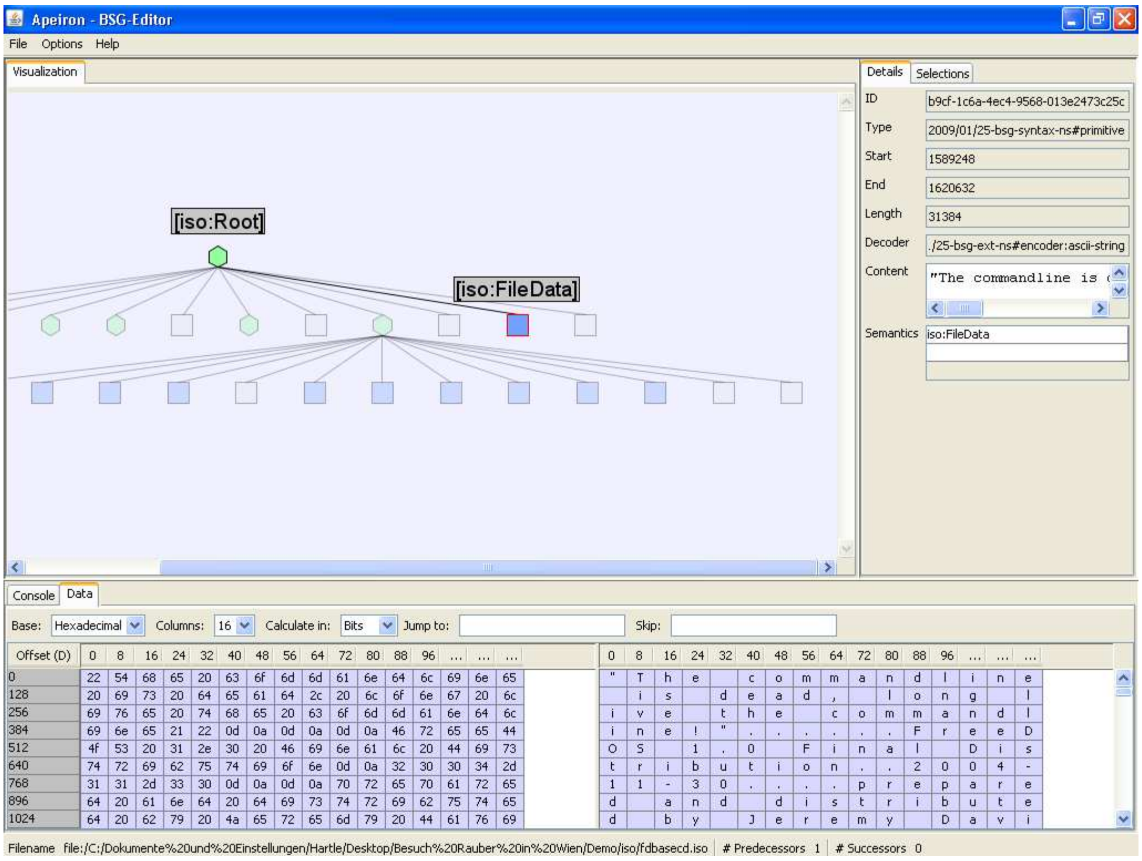
4.4.3 Providing tool support with the Apeiron BSG Editor

The Apeiron BSG Editor shown in Figure 4.11 uses the interactive visual representation (top left) with a hexdump representation (bottom) and a detailed property view for selected bitstream segments (right) to enable manual construction and modification of BSG instances using operations, and to enable the measurement of its descriptive completeness. Loading and saving BSG instances from and to files uses the RDF/N3-based digital storage representation previously defined.

The editor has been implemented in Java and uses OSGi and Eclipse Equinox for modularisation, so extending and modifying the editor, for example for an additional pushdown / pullup operation, can be done through OSGi bundles. The graph-based visualisation of BSG instances is based on the Prefuse visualisation library, and both storage and modification of the in-memory RDF graph representation make use of the Sesame and Elmo libraries. The implementation used for documenting these results are based on the the master thesis of Marcus Ständer, and on the diploma thesis of Friedrich-Daniel Möller, who realised a precursor version.

4.5 Applications of the BSG model

Describing data format instances using the BSG model enables applications in several domains, such as IT Security or Digital Preservation. In this thesis, two applications are given as examples, namely a *description of the PNG raster image "litmus test"* and the *documentation of two exploits* in IT Security. Other potential applications include *reverse-engineering of legacy data formats* for documentation in Digital Preservation through data format registries.



ID: b9cf-1c6a-4ec4-9568-013e2473c25c
Type: 2009/01/25-bsg-syntax-ns#primitive
Start: 1589248
End: 1620632
Length: 31384
Decoder: ./25-bsg-ext-ns#encoder:ascii-string
Content: "The commandline is
Semantics: iso:FileData

Base: Hexadecimal Columns: 16 Calculate in: Bits Jump to: Skip:

```
Offset (D)  0  8 16 24 32 40 48 56 64 72 80 88 96 ... ... ...    0  8 16 24 32 40 48 56 64 72 80 88 96 ... ... ...
0          22 54 68 65 20 63 6f 6d 6d 61 6e 64 6c 69 6e 65       "  T  h  e     c  o  m  m  a  n  d  l  i  n  e
128        20 69 73 20 64 65 61 64 2c 20 6c 6f 6e 67 20 6c          i  s     d  e  a  d  ,     l  o  n  g     l
256        69 76 65 20 74 68 65 20 63 6f 6d 6d 61 6e 64 6c       i  v  e     t  h  e     c  o  m  m  a  n  d  l
384        69 6e 65 21 22 0d 0a 0d 0a 0d 0a 46 72 65 65 44       i  n  e  !  "  .  .  .  .  .  .  F  r  e  e  D
512        4f 53 20 31 2e 30 20 46 69 6e 61 6c 20 44 69 73       O  S     1  .  0     F  i  n  a  l     D  i  s
640        74 72 69 62 75 74 69 6f 6e 0d 0a 32 30 30 34 2d       t  r  i  b  u  t  i  o  n  .  .  2  0  0  4  -
768        31 31 2d 33 30 0d 0a 0d 0a 70 72 65 70 61 72 65       1  1  -  3  0  .  .  .  .  p  r  e  p  a  r  e
896        64 20 61 6e 64 20 64 69 73 74 72 69 62 75 74 65       d     a  n  d     d  i  s  t  r  i  b  u  t  e
1024       64 20 62 79 20 4a 65 72 65 6d 79 20 44 61 76 69       d     b  y     J  e  r  e  m  y     D  a  v  i
```

Filename file:/C:/Dokumente%20und%20Einstellungen/Hartle/Desktop/Besuch%20Rauber%20in%20Wien/Demo/iso/fdbasecd.iso | # Predecessors 1 | # Successors 0

Figure 4.11: The Apeiron BSG Editor with a BSG instance for the FreeDOS Base CD ISO image file, highlighting a bitstream segment containing the contents of a text file.

4.5.1 Description of the PNG raster image "litmus test"

A suitable data format instance is the Portable Network Graphics (PNG) raster image file "oi2n0g16.png". The file consists of a structure starting with a signature, followed by a number of so-called chunks. The signature is a primitive carrying a fixed bit sequence for identifying the file as a PNG image. The chunks are again structures consisting of a length identifier as primitive, a type identifier as primitive, a type-dependent data segment and a CRC determinant as primitive. Depending on the type, chunks carry different data, such as the image header in the "IHDR" chunk, or image data in one or more "IDAT" chunks. In the given example, the image data is fragmented over two IDAT chunks, which needs to be composed first. The composed data does not directly correspond to the actual image, but is a transcode that is compressed using the GZIP algorithm. The uncompressed data is again a transcode where pixel data is rearranged to improve the GZIP compression efficiency. The reordered data then finally corresponds to the image data that represents the actual pixels with their given colour.

For this example, the corresponding BSG instance is presented in Figure 4.12. The BSG instance contains structures, primitives, fragments, a composite and two transcodes. It uses two block transformations for the GZIP compression and the

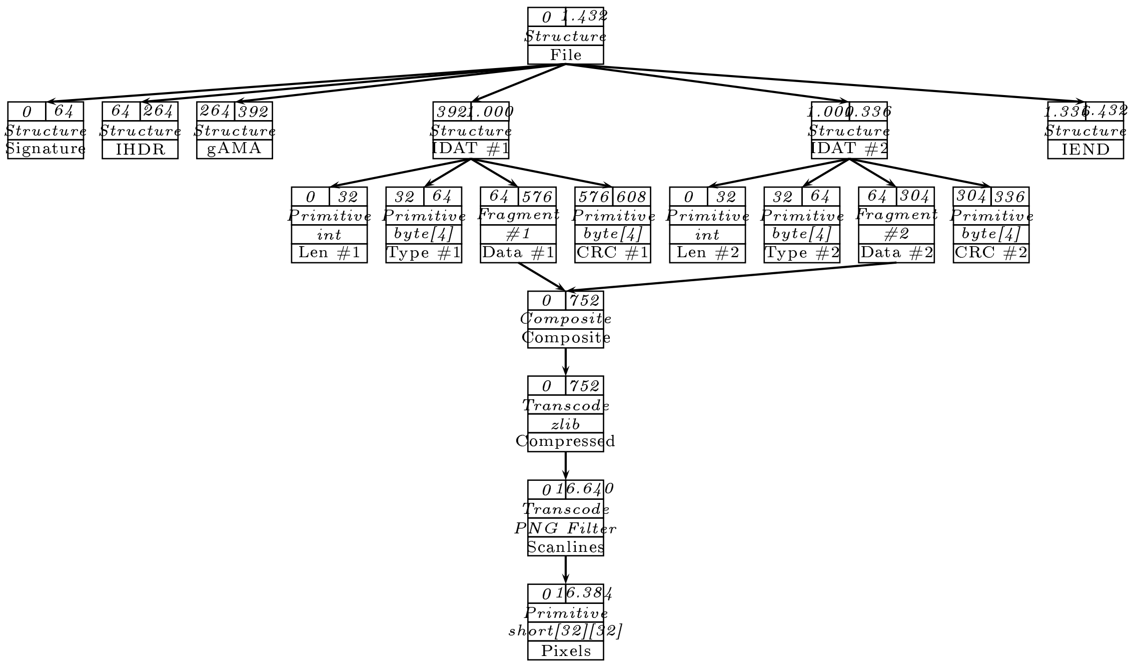


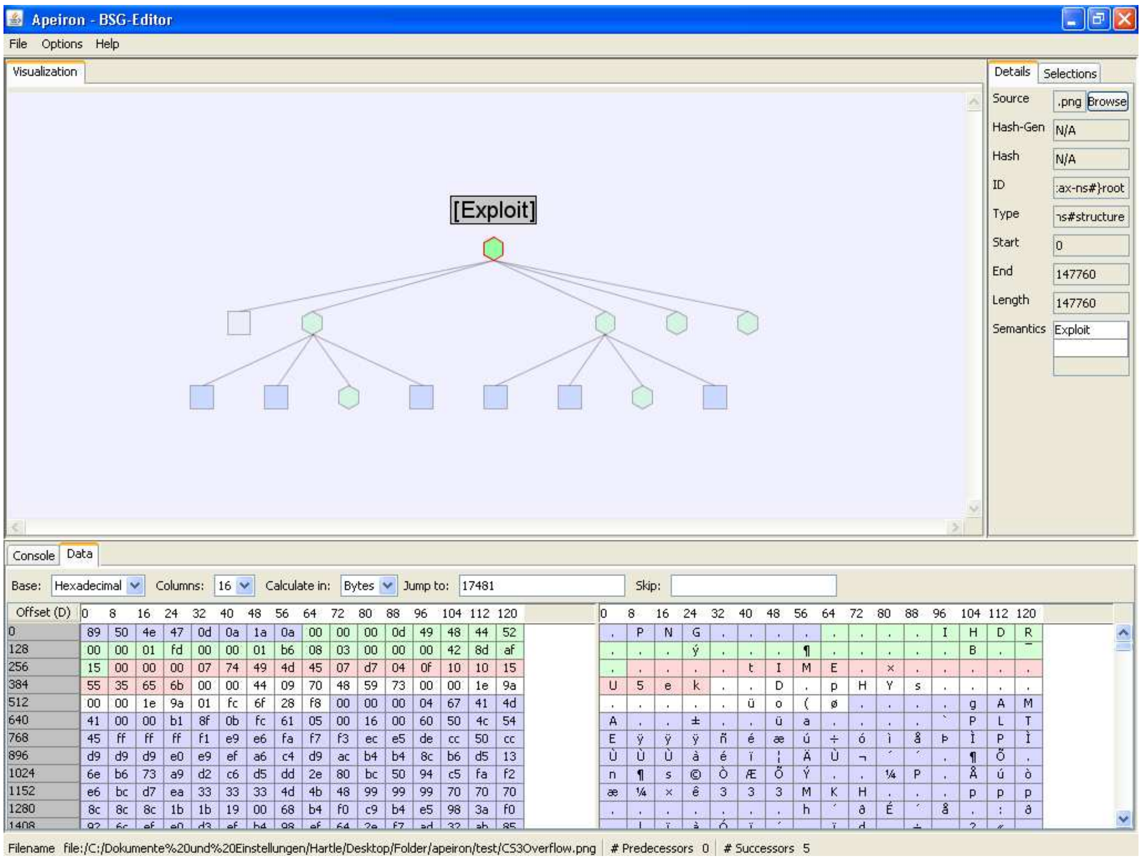
Figure 4.12: Partial bitstream segment graph for file "oi2n0g16.png", showing the bijective mapping of two PNG IDAT chunks to a 16 bit grayscale image with a resolution of 32×32 pixel, using the extended visual representation.

Scanline transformation, and two codings for integer and ASCII string primitives. Moreover, it shows a number of functional dependencies, such as a length determinant for the chunk length, a type determinant for the chunk type and a functional dependency on segment values for the CRC. The given BSG instance thus exercises all elementary descriptive capabilities that were previously identified in Section 3.4.3 and are provided by the BSG model.

4.5.2 Describing exploits in IT Security

An application of Bitstream Segment Graphs is the structural description of so-called exploits in IT Security. In order to inject and execute malicious code in an application, an exploit addresses its vulnerabilities, specific implementation errors in an application related to the processing of data. A well-known vulnerability in a number of image-processing tools from Adobe and Corel on the Windows platform is CVE-2007-2365, which is triggered by loading a crafted PNG image as exploit. For this vulnerability, there exists an exploit generator, which provides the option to chose from two different payloads, where one launches the Windows Calculator, while the other binds a shell to TCP port 4444, enabling local access to a remote attacker.

In this example, the former variant of a generated exploit is annotated. Moreover, this exploit is specifically considered with regards to its effects on Adobe Photoshop CS2. For annotating the exploit using the Apeiron tool, both the PNG W3C specification [Duc03] as well as the publicly available source code of the exploit generator in the programming language C [Mar07] are used.



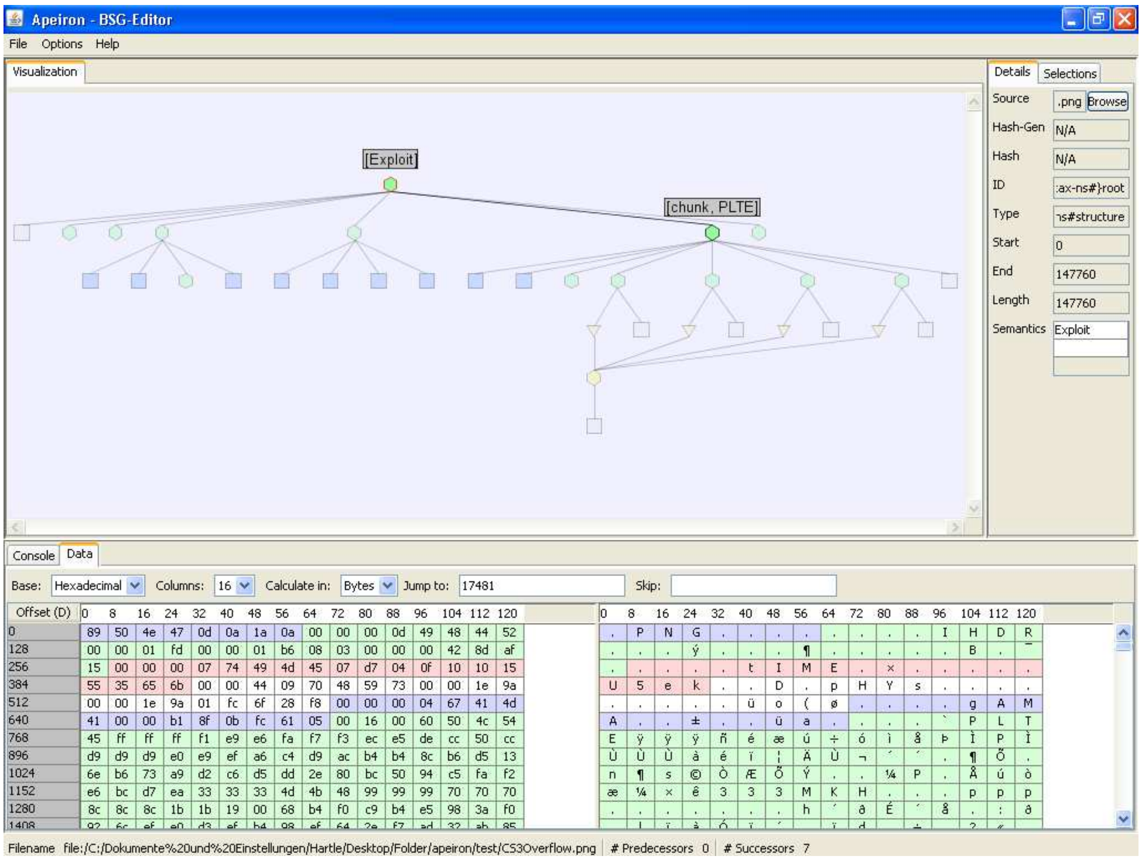
```
Source     .png  Browse
Hash-Gen   N/A
Hash       N/A
ID         :ax-ns#}root
Type       ns#structure
Start      0
End        147760
Length     147760
Semantics  Exploit

Base: Hexadecimal   Columns: 16   Calculate in: Bytes   Jump to: 17481   Skip:

Offset (D) 0  8  16 24 32 40 48 56 64 72 80 88 96 104 112 120
0          89 50 4e 47 0d 0a 1a 0a 00 00 00 0d 49 48 44 52
128        00 00 01 fd 00 00 01 b6 08 03 00 00 00 42 8d af
256        15 00 00 00 07 74 49 4d 45 07 d7 04 0f 10 10 15
384        55 35 65 6b 00 00 44 09 70 48 59 73 00 00 1e 9a
512        00 00 1e 9a 01 fc 6f 28 f8 00 00 00 04 67 41 4d
640        41 00 00 b1 8f 0b fc 61 05 00 16 00 60 50 4c 54
768        45 ff ff ff f1 e9 e6 fa f7 f3 ec e5 de cc 50 cc
896        d9 d9 d9 e0 e9 ef a6 c4 d9 ac b4 b4 8c b6 d5 13
1024       6e b6 73 a9 d2 c6 d5 dd 2e 80 bc 50 94 c5 fa f2
1152       e6 bc d7 ea 33 33 33 4d 4b 48 99 99 99 70 70 70
1280       8c 8c 8c 1b 1b 19 00 68 b4 f0 c9 b4 e5 98 3a f0

0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 120
. P N G . . . . . . . . I H D R
. . . ý . . . ¶ . . . . . B . ¯
. . . . . t I M E . × . . . . .
U 5 e k . . D . p H Y s . . . .
. . . . . ü o ( ø . . . . g A M
A . . ± . . ü a . . . . ` P L T
E ÿ ÿ ÿ ñ é æ ú ÷ ó ì å Þ Ì P Ì
Ù Ù Ù à é ï ¦ Ä Ù ¬ ´ ´ . ¶ Õ .
n ¶ s © Ò Æ Õ Ý . . ¼ P . Å ú ò
æ ¼ × ê 3 3 3 M K H . . . p p p
. . . . . . . h ´ ð É ´ å . : ð

Filename file:/C:/Dokumente%20und%20Einstellungen/Hartle/Desktop/Folder/apeiron/test/CS3Overflow.png | # Predecessors 0 | # Successors 5
```

Figure 4.13: The structure of the crafted PNG image exploit shown in Apeiron, including a PNG signature, an "IHDR" chunk, a "tIME" chunk and an invalid "pYHs" chunk, followed by unknown data in a generic segment.

Solely considering the exploit as a PNG image according to the PNG file format specification, a partial BSG instance can be constructed as shown in Figure 4.13. The partial instance describes the layout of a PNG image signature and three PNG *chunks*, where each PNG chunk consists of a 32-bit length descriptor, a 32-bit type descriptor, variable-length data with a structure depending on the type of chunk, and a 32-bit CRC value on both the type and data field, where its length is non-zero. These three chunks are:

- an "IHDR" chunk, which covers basic information regarding the image itself (a 509 pixels × 438 pixels resolution with 256 indexed colours from a palette, and standard values regarding PNG image compression, filtering and interlacing), followed by
- a "tIME" chunk, which covers the last modification date of the image (on 2007-04-15 on 16:16:21 o'clock), and finally followed by
- a "pHYs" chunk, which covers the physical dimensions of the image pixels.

While both the "IHDR" and "tIME" chunks follow the PNG file format specification, the "pHYs" chunk is invalid, as it is required to have a value of 0x9 for



Source	.png
Hash-Gen	N/A
Hash	N/A
ID	:ax-ns#}root
Type	ns#structure
Start	0
End	147760
Length	147760
Semantics	Exploit

Base: Hexadecimal Columns: 16 Calculate in: Bytes Jump to: 17481 Skip:

```
Offset (D) 0  8  16 24 32 40 48 56 64 72 80 88 96 104 112 120
0          89 50 4e 47 0d 0a 1a 0a 00 00 00 0d 49 48  44  52
128        00 00 01 fd 00 00 01 b6 08 03 00 00 00 42  8d  af
256        15 00 00 00 07 74 49 4d 45 07 d7 04 0f 10  10  15
384        55 35 65 6b 00 00 44 09 70 48 59 73 00 00  1e  9a
512        00 00 1e 9a 01 fc 6f 28 f8 00 00 00 04 67  41  4d
640        41 00 00 b1 8f 0b fc 61 05 00 16 00 60 50  4c  54
768        45 ff ff ff f1 e9 e6 fa f7 f3 ec e5 de cc  50  cc
896        d9 d9 d9 e0 e9 ef a6 c4 d9 ac b4 b4 8c b6  d5  13
1024       6e b6 73 a9 d2 c6 d5 dd 2e 80 bc 50 94 c5  fa  f2
1152       e6 bc d7 ea 33 33 33 4d 4b 48 99 99 99 70  70  70
1280       8c 8c 8c 1b 1b 19 00 68 b4 f0 c9 b4 e5 98  3a  f0
```

```
0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 120
. P N G . . . . . . . . I H D R
. . . ý . . . ¶ . . . . . B . ¯
. . . . . t I M E . × . . . . .
U 5 e k . . D . p H Y s . . . .
. . . . . ü o ( ø . . . . g A M
A . . ± . . ü a . . . . ` P L T
E ÿ ÿ ÿ ñ é æ ú ÷ ó ì å Þ Ì P Ì
Ù Ù Ù à é ï ¦ Ä Ù ¬ ´ ´ . ¶ Õ .
n ¶ s © Ò Æ Õ Ý . . ¼ P . Å ú ò
æ ¼ × ê 3 3 3 M K H . . . p p p
. . . . . . . h ´ ð É ´ å . : ð
```

Filename file:/C:/Dokumente%20und%20Einstellungen/Hartle/Desktop/Folder/apeiron/test/CS3Overflow.png | # Predecessors 0 | # Successors 7

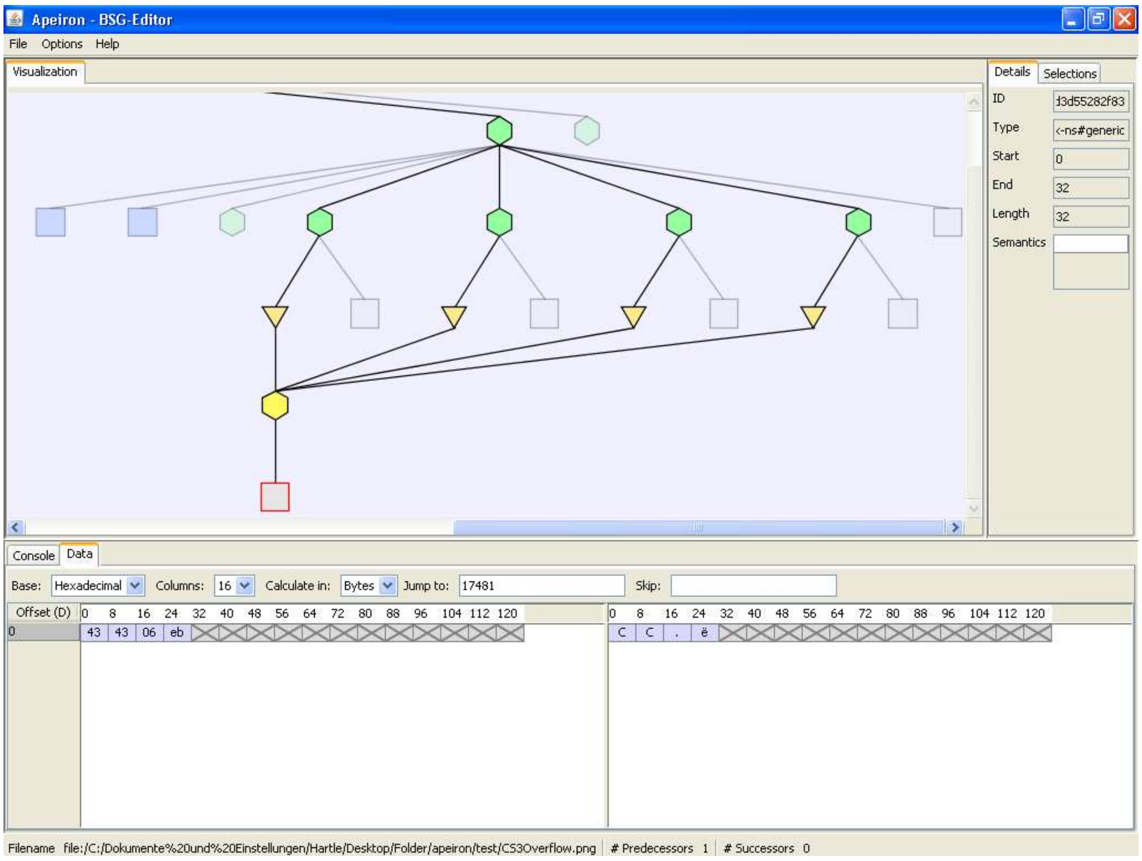
Figure 4.14: The structure of the crafted PNG image exploit shown in Apeiron extended from Figure 4.13, additionally including a “gAMA” chunk and an invalid “PLTE” chunk when adjusting the previously invalid “pYHs” length descriptor.

its length descriptor, yet it has a value of 0x4409. Even when assuming this chunk to be valid, the hypothetically following PNG chunk then would have a value of 0xb67d641e for its length descriptor, by far exceeding the length of the overall image file. The invalid "pHYs" chunk is therefore followed by a generic segment with data of so-far unknown purpose.

Either by masking the value of the length descriptor of the invalid "pHYs" chunk to the least significant byte, or by ignoring the actual value and instead considering its value as defined by the specification, it can be observed that the chunk is followed by two further chunks as shown in Figure 4.14:

- a "gAMA" chunk, which covers the gamma value of the image, and
- a "PLTE" chunk, which describes the indexed colours of the image palette.

While the "gAMA" chunk follows the PNG specification, the "PLTE" chunk is again invalid. Each of its 256 colours should be defined through the primary colours red, green and blue, with 1 byte for each colour component. As such, its length descriptor is required to have a value of 0x300, yet it has a value of 0x160060, which again exceeds the length of the image file.


Apeiron - BSG-Editor

File Options Help

Visualization

Details Selections

ID d3d55282f83
Type <-ns#generic
Start 0
End 32
Length 32
Semantics

Console Data

Base: Hexadecimal Columns: 16 Calculate in: Bytes Jump to: 17481 Skip:

```
Offset (D) 0  8  16 24 32 40 48 56 64 72 80 88 96 104 112 120   0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 120
0          43 43 06 eb                                          C C .  ë
```

Filename file:/C:/Dokumente%20und%20Einstellungen/Hartle/Desktop/Folder/apeiron/test/CS3Overflow.png | # Predecessors 1 | # Successors 0

Figure 4.15: The structure of the crafted PNG image exploit shown in Apeiron extended from Figure 4.14, additionally including the first four bytes of fragmented shellcode from the red colour component in indexed colours 1496 and up.

Examing the source code of the exploit generator, the embedded shellcode is embedded into the "PLTE" chunk by fragmenting it into separate bytes every three bytes, which corresponds to the red colour component of the indexed colours 1496 and following. Figure 4.15 shows the partial BSG instance including a concatenation of the first four shellcode bytes.

In this form, the embedded shellcode depends on its reassembly through the targeted application in order to be executed. It is therefore reasonable to assume that its exploit triggers a buffer overflow of a buffer intended for the red colour component of an indexed colour palette, which would conveniently provide for the required reassembly of its shellcode.

The partial BSG instance annotated so far thus provides the location of its embedded shellcode and helps to give a clue as to how its execution is triggered. The example shows how this approach allows a security export to systematically decompose malicious data for purposes of documentation and further analysis.

4.6 Discussion

The BSG model has a number of properties that distinguish it from the set of examined related work. The model presented in this chapter provides *completeness of descriptive capabilities*, enables the *explicit mapping between an external representation and a canonical internal representation* and supports the *incremental construction and modification* of data format instances. At the same time, the BSG model offers a *conceptual simplicity* not present in examined related work, and enables a *representation of data format instances suitable for humans and machines*:

- **Completeness of descriptive capabilities:** In contrast to examined related work, the BSG model is complete in its descriptive capabilities, as it is based on the previously presented abstraction of a rooted causality graph, and thus provides all four elementary descriptive capabilities of decoding primitive data, segmenting structured data, transforming transcoded data and concatenating fragmented data. It supports descriptions at arbitrary levels of granularity down to individual bits.

- **Explicit mapping between an external representation and a canonical internal representation:** Compared to examined related work, a BSG instance makes the way certain information is actually represented explicit. Where other approaches only provide an XML-based or similar representation corresponding to an external representation, the BSG model makes the overall mapping explicit, which can be helpful in verifying the way an application processes external representations, allowing a comparison of the expected and actual mapping of information in a data format instance.

- **Incremental construction and modification:** The BSG model supports the incremental construction of its instances. It is reasonable to assume that the complexity of describing the composition of data is reduced when a model provides support for incremental construction and modification, so the BSG model allows for incomplete descriptions to be completed in incremental steps by several participants.

- **Conceptual simplicity:** Compared to the complexity of other approaches on describing the composition of data, the BSG model is conceptually simple. Based on the analysis in Chapter 3, its realisation in the BSG model results to a conceptually simple model, where there are only 6 different types of data, namely primitive data, structured data, transcoded data, fragmented data, concatenated data and generic data. Through only 8 pairs of well-defined operations and their respective inverse, it is possible to construct and modify arbitrary BSG model instances, given the existence of required bitstream coding and bitstream transformation functions.

- **Representation of data format instances suitable for humans and machines:** The BSG model provides a representation that is suitable for sharing between humans and machines, as it can be visualised graphically and operated on by humans as well as processed automatically due to its RDF-based representation.

Furthermore, two remaining aspects need to be discussed, such as the *separation of the causality graph and the mapping step implementations*, and the observation of *redundancy of information in data format instances*:

- **Separation of the causality graph and the mapping step implementations:** Although a data format instance may involve computationally complex mapping steps, the intention is to describe the composition of data as specific as possible, rather than using arbitrary universal computational devices.

  Encodings of data types and block transformations of data are of interest, yet their number is small compared to the growing number of data formats in use, as observed in Digital Preservation [RH05]. Both bitstream coding and transformation functions can be factored out as reversible computational devices. These can then be standardised and reused in canonical descriptions of data formats.

  Separating the actual implementation of coding and transformation functions from the description allows for different realisations of implementations, the standardisation of block transformations and codings regarding their specification, and keeps the model for describing data format instances focused. While the bijectivity of these functions can be ensured through reversible computational devices such as RTMs, actually implementing these functions using reversible computational devices requires further research on reversible high-level programming languages for the definition of bitstream coding and bitstream transformation functions.

- **Redundancy of information in data format instances:** It can be observed that sometimes there are several different paths to obtain a certain piece of information. For example, the length of a segment can be inferred from neighbouring segments or through separate packaging segments which carry the desired length information explicitly. When the information shall be inferred in an automated fashion, there may be cylic paths of reasoning which do not terminate in a depth-first search. Rather than that, it is necessary to perform a width-first search, trying to reach all possible facts, but terminating when there are no more new facts to be reached. This leads to the idea of computing a BSG instance description as a fixed-point through reasoning.

4.7 Summary

This chapter defines the Bitstream Segment Graph (BSG) model for describing arbitrary data format instances, which realises the formalisation introduced in the previous Chapter 3. A number of visual representations as well as an RDF-based storage representation have been given, enabling the storage and exchange of BSG model instances. In order to construct and modify BSG instances, a closed set of 8 pairs of operations and their inverses are given and have been implemented in the Apeiron BSG Editor as tool support. Its capabilities have been demonstrated through a description of the PNG image “litmus test” as well as through a description of a PNG raster image file which is a known exploit that targets errors in certain image processing tools such as Adobe Photoshop CS2, thereby triggering the execution of an embedded malicious payload. In a discussion, a number of properties are pointed out which distinguish the BSG model from related work previously examined in Chapter 2, such as its completeness of descriptive capabilities or its conceptual simplicity.

Chapter 5

Describing Data Formats

5.1 Introduction

The previous Chapter 4 presented the Bitstream Segment Graph model for describing the composition of data through a causality graph. Building on the BSG model, this chapter presents the BSG Reasoning approach, which enables the description of data formats through logic-based rules, addressing the following aspects:

- **Definition of the BSG Reasoning approach:** This chapter presents the *BSG Reasoning* approach for describing a potentially infinite set of data format instances in Section 5.2.
- **Representation of BSG Reasoning rulesets:** Using the previously introduced approach, the representation of rulesets is defined in Section 5.3.
- **Applications of BSG Reasoning:** The presented approach is evaluated in Section 5.4 by describing a set of Portable Network Graphics (PNG) image files using a subset of the PNG image file format that covers all elementary descriptive capabilities. Furthermore, another application for the BSG Reasoning approach is in IT Security with format-aware fuzzing of bitstreams is outlined.

5.2 Definition of the BSG Reasoning aproach

Based on established theoretical foundations in logic [Hed04], the following definitions can be introduced.

5.2.1 Making propositions

In order to express data format rules, a number of definitions need to be introduced first.

Definition 5.2.1 (Literal): A *literal* is a fixed value such as the string `'x'` or the number constant `1`.

Definition 5.2.2 (Variable): A *variable* is an identifier such as `y` that may serve as placeholder for a literal. A variable can either be *bound* to a literal, thus representing it, or be *free*. For a bound variable, there exists a *binding* of the variable to a literal. For a free variable, no binding is defined.

Example 5.2.1: Given the variables `x` and `y` as well as a variable binding $x = 1$, then the variable `x` is bound, whereas `y` is not.

Definition 5.2.3 (Term): A *term* is either a literal or a variable.

Definition 5.2.4 (Predicate): A *predicate* defines a relation and is denoted `name/arity`, where the *name* designates the relation and where the *arity* is the number of terms for a tuple in the relation. A *predicate function* takes a number of terms $x_0, \ldots, x_n$ as arguments and decides membership of the tuple $t = (x_0, \ldots, x_n)$ in its predicate relation.

Example 5.2.2: The predicate `math:sum/3` has a predicate function $f(x_0, x_1, x_2)$ which decides the membership of $t = (x_0, x_1, x_2)$ in its relation. The predicate function f tests whether $x_0 + x_1 = x_2$ holds.

Example 5.2.3: The predicate `person:related-to/2` has a predicate function $f(x_0, x_1)$ which decides the membership of $t = (x_0, x_1)$ in its relation. The predicate function f tests whether a person x_0 is related to person x_1.

Definition 5.2.5 (Proposition): A *proposition* proposes a statement as fact, which is expressed as predicate function with a number of terms as its argument. A proposition is *ground* if all terms are literals or bound variables. A proposition is said to *hold* if the predicate function decides its membership positively.

Example 5.2.4: `math:sum(1, 4, 5)`, `math:sum(1, 4, 8)` and `math:sum(1, 4, c)` are all propositions. Given no variable binding for the variable `c`, the first and second propositions are ground, whereas the third proposition is not. Given the variable binding $c = 5$, both the first and the third proposition hold, whereas the second does not.

Example 5.2.5: Assuming Jack to be related to Gill, and thus $(Jack, Gill)$ to be in the relation defined by the predicate `person:related-to/2`, then the proposition `person:related-to('Jack', 'Gill')` holds.

5.2.2 Using predicates

The relation defined by a predicate can either be *open* or *closed*, leading towards the *Open World Assumption* and *Closed World Assumption*:

DEFINITION 5.2.6 (OPEN WORLD ASSUMPTION): Under the *Open World Assumption (OWA)*, a proposition not known to be true may become true when further knowledge becomes available at a later point in time.

DEFINITION 5.2.7 (CLOSED WORLD ASSUMPTION): Under the *Closed World Assumption (CWA)*, a proposition not known to be true is known to be false, regardless of further knowledge becoming available or not.

Computable and inferred predicates

Both OWA and CWA are required for the next two definitions, where a predicate is either *computable* or *inferred*:

DEFINITION 5.2.8 (COMPUTABLE PREDICATES): A computable predicate describes a closed set of propositions following the CWA.

EXAMPLE 5.2.6: The predicate `math:sum/3` used in previous examples is computable and follows the CWA. As the proposition `math:sum(1, 4, 8)` does not hold at any step during the inference process, it will not hold at any later step as well. Among other computable predicates, the predicate `math:sum/3` will later be used for data format description in order to compute the start position, length and end position of bitstream segments, or to validate their consistency.

DEFINITION 5.2.9 (INFERRED PREDICATE): An inferred predicate describes an open set of propositions following the OWA.

EXAMPLE 5.2.7: The predicate `person:related-to/2` used in previous examples is inferred and follows the OWA. While the proposition `person:related-to('John', 'Betty')` does not hold now, the proposition may still be inferred and thus hold later. A number of inferred predicates will later be used for data format description to infer properties and relations between bitstream segments, such as the start position of a bitstream segment, or neighbourship relations between two adjacent bitstream segments.

Computable and inferable predicates thus differ in the semantics of the return value of their predicate function. For computable predicates with propositions that do not hold, the negative return value is authoritative. For inferred predicates with propositions that do not hold, the negative return value is authoritative only for the current state of knowledge, and thus may change with future knowledge.

As defined so far, ground propositions of predicates can be evaluated. For evaluating propositions with free variables, further definitions are required.

DEFINITION 5.2.10 (MODE): Given a predicate `name/arity` with its predicate function f, a *mode* is a tuple $t = (b_1, \ldots, b_{arity})$, where each element $b_n \in \{true, false\}$ states whether f supports a free variable at argument position n.

Every predicate has at least its *base mode* $(b_1, \ldots, b_{arity})$ with $b_n = false$, where it does not support free variables at any argument position. If a predicate supports further modes, then there exists a function which can finitely enumerate tuples of potential variable bindings for which the proposition becomes ground and holds.

Where computable predicates define additional modes at their discretion, inferable predicates support arbitrary modes other than the base mode, as these can always finitely enumerate their known potential variable bindings for its propositions that hold.

Example 5.2.8: For the predicate `math:sum/3`, functions can be defined to test ground propositions such as `math:sum(1,2,3)` and to finitely enumerate variable bindings for propositions such as `math:sum(a,4,5)`, `math:sum(1,b,5)` as well as `math:sum(1,4,c)`. This predicate therefore supports the modes $(false, false, false)$, $(true, false, false)$, $(false, true, false)$ and $(false, false, true)$. The finite enumeration of variable bindings in the latter three modes yield the variable bindings $(a = 1)$, $(b = 4)$ and $(c = 5)$. Applied to their respective proposition, they all produce the same ground proposition `math:sum(1, 4, 5)`.

Example 5.2.9: As the predicate `person:related-to/2` is inferred, it supports arbitrary modes besides the base mode. Given the previously stated relations and the propositions `person:related-to('John', b)`, `person:related-to(a, 'Betty')` and `person:related-to(a, b)`, the finite enumeration of variable bindings yields $(b =' Gill')$, the empty set $\varnothing$, and $(a =' John', b =' Gill')$. For the first and last proposition, applying the enumeration to their respective proposition leads to the same ground proposition `person:related-to('John', 'Gill')` which holds. For the second proposition, no ground proposition is obtained with the current knowledge.

For using the BSG Reasoning approach, predicates have been defined which *represent properties of the BSG model*, *perform mathematical operations* or *provide utility functions*.

BSG-related predicates

To address properties of a BSG model instance, the following inferable predicates have been defined:

- `bsg:source(a, b)`: This predicate states that a bitstream segment a is a bitstream source as defined by the file reference b.
- `bsg:start(a, b)`: This predicate states that a bitstream segment a starts at bit position b (inclusive) within its predecessor(s).
- `bsg:length(a, b)`: This predicate states that a bitstream segment a has the length b in bits.
- `bsg:end(a, b)`: This predicate states that a bitstream segment a ends at bit position b (exclusive) within its predecessor(s).
- `bsg:leads(a, b)`: This predicate states that a bitstream segment a leads a neighbouring bitstream segment b, so the end position of a is the start position of b.

- `bsg:follows(a, b)`: This predicate states that a bitstream segment a follows a neighbouring bitstream segment b, so the start position of a is the end position of b. This predicate is the inverse of the previous `bsg:lead/2` predicate.
- `bsg:successor(a, b)`: This predicate states that a bitstream segment a has a bitstream segment b as its successor.
- `bsg:firstSuccessor(a, b)`: This predicate states that a bitstream segment a has the bitstream segment b as its first successor.
- `bsg:lastSuccessor(a, b)`: This predicate states that a bitstream segment a has the bitstream segment b as its last successor.
- `bsg:type(a, b)`: This predicate states that a bitstream segment a has the type b, which may be the string constants `'generic'`, `'primitive'`, `'structure'`, `'transcode'`, `'fragment'` or `'composite'`.
- `bsg:resolved(a)`: This predicate states whether a bitstream segment a has been *resolved*, that is, whether its start, length, end, type and predecessor are known, so that it is possible to resolve the bit sequence of this bitstream segment through a path to the (resolved) root bitstream segment.
- `bsg:semantics(a, b)`: This predicate states specifically assigned semantics of a bitstream segment a in terms of a string b. For a defined bitstream segment a, there may be multiple strings b present.
- `bsg:encoding(a, b)`: This predicate states that a primitive bitstream segment a uses the bitstream coding function b, which is a string reference identifying the actual function.
- `bsg:transcoding(a, b)`: This predicate states that a transcode bitstream segment a uses the bitstream transformation function b, which is a string reference identifying the actual transformation.

Mathematical predicates

For performing mathematical operations, the following computable predicates have been defined:

- `math:lt(a, b)`: This predicate states that $a < b$.
- `math:lte(a, b)`: This predicate states that $a \leq b$.
- `math:product(a, b, c)`: This predicate states that $a \times b = c$. If only one of a, b or c is a free variable, then its value is computed.
- `math:sum(a, b, c)`: This predicate states that $a + b = c$. If only one of a, b or c is a free variable, then its value is computed.

Utility predicates

For additional utility functions, the following computable predicates have been defined:

- `util:concat(a, b, c)`: This predicate states that the string concatenation of a and b results in c. If only c is a free variable, then this predicate concatenates the strings a and b.

- `util:sourceLength(a, b)`: This predicate states that a source reference a of a bitstream source has the length b in bits. If a is a bound variable or a term representing a string which contains a reference to a file, then its size in bits is obtained. If b is a term representing a number, then equality its value and the source size in bits is tested. If b is a free variable, then it is bound to the size of the source in bits.

- `util:skolem(`x_0`, `x_1`, ..., `x_n`)`: This predicate states that for a rule identifier x_0 and a number of terms from universally quantified variables in $x_1 \dots x_{n-1}$, there exists a value x_n. If only x_n is a free variable, then a unique identifier for $x_0 \dots x_{n-1}$ is obtained and bound. This predicate is introduced by rule transformations for skolemization, and thus not used directly for specification.

- `util:value(a, b)`: This predicate states that a primitive bitstream segment a has a decoded value of b. If a is a bound variable representing a bitstream segment which is a primitive and is resolved, then its value is decoded. If b is a bound variable or a term, then its equivalence is tested against the decoded value. If b is a free variable, then it is bound to the decoded value.

- `util:crc(a, b)`: This predicate states that a sequence of bytes a has a Cyclic Redundancy Code (CRC) of b. If a is a bound variable, then its CRC is computed. If b is a bound variable or a term, then its equivalence to the computed CRC is tested, otherwise the computed CRC is bound.

5.2.3 Defining rules

In order to express complex statements using multiple predicates, some means for logic compositions of propositions is required.

Definition 5.2.11 (Logic operators): The operators $\wedge$, $\vee$ and $\neg$ represent the binary 'and' and 'or' logic operators as well as the unary 'not' logic operators to be used for propositions.

Example 5.2.10: The proposition $\neg$ `math:sum(1, 4, 8)` holds, as the proposition `math:sum(1, 4, 8)` does not hold.

Definition 5.2.12 (Formula): A formula consists of one or more propositions connected by logic operators. An *atomic* formula consists of a single proposition which may be negated. A formula is *closed* if all of its propositions are ground and *open* otherwise. A formula *holds* if the corresponding logic statement on its proposition holds as well.

Example 5.2.11: The formula `math:sum(1, 4, a)` $\wedge$ `math:sum(2, 3, a)` holds for the variable binding $a = 5$. Both propositions use the mode $(false, false, true)$ of predicate `math:sum/3`, so their variable bindings can be finitely enumerated, yielding $a = 5$. Applying this variable binding $a = 5$ to all propositions in the formula, these become ground, so the formula becomes closed and holds.

In the use of variables, distinguishing between different forms of quantification is needed.

Definition 5.2.13 (Existential and universal quantifiers): The operators $\exists$ and $\forall$ represent the existential and universal quantifiers for variables. By default, variables are considered as universally quantified unless stated otherwise.

Based on quantified formulas, the concept of a *rule* can finally be defined:

Definition 5.2.14 (Rule): A rule is an implication $a \Rightarrow b$ consisting of a and b as formulas which denote the *condition* a and its *conclusion* b. Variables are quantified as existential or universal over the entire rule. A rule *matches* if there exists a set of variable bindings for which the condition formula is closed and holds. If a rule matches for a given set of variable bindings, the conclusion formula is closed by applying the variable bindings, testing for contradictions and inferring new propositions. A *contradiction* in a rule is a computable predicate in a conclusion formula which does not hold. If there is no contradiction, then all propositions on inferable predicates are inferred, with the respective relations updated.

In the context of the BSG model, rules can be classified as either *model-specific* or *format-specific*.

Definition 5.2.15 (Model-specific rules): A *model-specific* rule captures part of the BSG model itself and is independent from the specific data format to be described. Examples of such rules concern the consistency of start, length and end positions of a bitstream segment, or the consecutive placement of neighbouring child segments in a structure segment.

Example 5.2.12: Assume a structure bitstream segment a with two primitive bitstream segments b and c as its first and last successors. Since b and c are the only successors of a, they are necessarily neighbours, so that the end position of b is the start position of b. Therefore, a corresponding rule captures a property that is specific to the BSG model rather than specific to a format.

Definition 5.2.16 (Format-specific rules): A *format-specific* rule captures part of the data format to be described and is independent from the BSG model. An example is a rule how a specific structure is segmented, as mandated by the data format itself.

Example 5.2.13: Assume a structure bitstream segment a which represents a PNG raster image, and has a structure bitstream segment b as its first successor, which represents the PNG signature, and thus has a length of 64 bits. A corresponding rule captures a property that is specific to the PNG raster image file format.

5.2.4 Matching rule conditions

Rules are converted to Conjunctive Normal Form (CNF), resulting in a prepared set of rules, and skolemized. In order to improve on the search space to be evaluated, the problem of testing the condition of a rule is considered as a graph matching problem. The objective is to try to minimise the number of tests in order to decide whether a rule condition matches or not.

Due to preparation of rules, the formula representing the rule conditions is in CNF, consisting of one or more propositions logically connected with the operator $\wedge$. For a closed formula, no matching is necessary, as all propositions are ground and can be tested.

To match an open formula as condition, all sets of variable bindings have to be identified for which all propositions in the formula hold.

Considering all propositions in formula as tuple $t_0 = (p_1, \ldots, p_{n_0})$ and an initial set of variable bindings $b_0 = \{\}$.

If there is a ground proposition which does not hold, the rule condition fails. If all ground propositions hold, then these can be dropped from consideration and they are removed from the tuple, resulting in $t_y = (p_1, \ldots, p_{n_y})$, consisting of non-ground propositions. For all propositions p_x as elements in t_y, an ordering o_x is computed.

Each proposition is tested to determine whether its predicate function supports the current mode of the predicate. If it does not, then more variables have to be bound prior to being able to test the proposition, so its ordering is $o_x =$. If the predicate function of a proposition supports the current mode, then o_x is set to the count of ground propositions it can finitely enumerate.

The elements of the tuple t_y are then brought in ascending order of o_x.

5.2.5 Inference process

The prepared set of rules is evaluated in steps using semi-naïve evaluation, making it feasible for large sets of rules and inferred propositions by substantially improving the efficiency of the inference process. By focusing on rules that also make use of knowledge inferred in the previous step, *semi-naïve evaluation* prevents the same conclusions from being reached over and over again.

The BSG Reasoning engine therefore categorises every proposition of inferred predicates into separate *knowledge bases* which may be *new knowledge kb_n*, *current knowledge kb_c* or *old knowledge kb_o*:

- *New knowledge kb_n* contains propositions that are inferred in the current step of inference. By separating these propositions into kb_n, the ordering of rules has no influcence over the inference process.

- *Current knowledge kb_c* contains propositions that either have been inferred in the previous step of inference, or that have been given initially. Due to semi-naïve evaluation, at least one proposition contained in kb_c has to be used when matching a rule in the inference step, potentially inferring new knowledge.

- *Old knowledge kb_o* contains all propositions that have been inferred in steps of inference prior to the previous step.

During a step of inference, the reasoning engine consults both kb_o and kb_c to infer new propositions for kb_n. After all rules have been processed, propositions from current knowledge are removed and added to old knowledge. Likewise, all propositions from new knowledge are removed and added to current knowledge. The evaluation is repeated until there is no proposition in new knowledge after processing all rules, reaching a least fixed point.

EXAMPLE 5.2.14: Assume a bitstream source *'root'* which references the file *test.png* as initially given facts. The referenced file itself has a size of 1000 bytes, therefore 8000 bits. Furthermore, assume a single rule. The rule has the condition that a bitstream source x references a file f (`bsg:source(x, f)`) which has a length of y in bits (`util:sourceLength(f, y)`). The conclusion of the rule, if it matches, is that the bitstream segment x starts at bit position 0 (`bsg:start(x, 0)`) and that a has the length of y in bits (`bsg:length(x, y)`).

Initially given facts are stored in kb_c, while both kb_o and kb_n are empty, and the inference process begins:

- The existing facts in both kb_c and kb_o allow the condition of the single rule to match, and all facts come from the current knowledge kb_c, so the constraint of semi-naïve evaluation to use at least one proposition from current knowledge is satisfied. For matching the rule, this results in the variable binding $(x =' root'), (f =' test.png'), (y = 8000)$, for which the conclusion of the rule can be inferred.

  As both `bsg:start/2` and `bsg:length/2` are inferred predicates, the propositions `bsg:start('root', 0)` and `bsg:length('root', 8000)` are added to the relations of both predicates in the knowledge base kb_n.

  After the ruleset consisting of just one rule has been applied, new knowledge has been inferred, as kb_n is non-empty. Thus, current knowledge in kb_c is moved to old knowledge in kb_o, then new knowledge kb_n is moved to current knowledge kb_c, and a new inference step begins.

- In this new step, matching the rule requires the use of at least one proposition that has been inferred in the previous step, which is not possible. Therefore, since no new knowledge has been inferred, kb_n is empty and the inference process reaches a least fixed point.

When a least fixed point has been reached during inference, tuples contained in the relations of BSG predicates are used for constructing a BSG instance. In contrast to the manual operations of the BSG model introduced in the previous Chapter 4, the instance may be invalid, and thus has to be tested and validated in structural terms. For example, a bitstream segment without an inferred type is considered as a generic bitstream segment, and it has to be tested whether every bitstream segment has a path to the root bitstream segment.

5.3 Representation of BSG Reasoning rulesets

For representing data format rules, the following representation is used. It combines the expressivity of XML Common Logic with a less verbose representation using brackets. The following forms of expression are defined:

- **(prefix** *ns ref***):** Given an identifier *ns* and a string *ref* containing a URL namespace, this defines *ns* as namespace prefix for *ref*. Prefixes are used for namespacing predicates with the same name, yet different semantics.
- **(ns:pred** $x_0 \ldots x_n$**):** Given a tuple of terms $t = (x_0, \ldots, x_n)$ and the predicate `ns:pred/n`, this states the logic expression of testing whether t is in the relation defined by the predicate `pred` of arity n in the namespace prefix `ns` that was previously defined.
- **(and** $a\,b$**):** Given two logic expressions a and b, this states the logic expression of $a \wedge b$; if both a and b hold, then $a \wedge b$ holds as well.
- **(or** $a\,b$**):** Given two logic expressions a and b, this states the logic expression of $a \vee b$; if a or b holds, then $a \vee b$ holds as well.
- **(not** a**):** Given a logic expression a, this states the logic expression of $\neg a$; if a does not hold, then $\neg a$ holds.
- **(implies** $a\,b$**):** Given two logic statements a and b, this states the implication $a \Rightarrow b$ as logic statement; if a holds, then b is implied to hold.
- **(iff** $a\,b$**):** Given two logic statements a and b, this states the implications $a \Rightarrow b$ and $b \Rightarrow a$ as logic statements; a holds exactly iff (**if** and only **if**) b holds.
- **(forall** $x_0 \ldots x_n\, e$**):** Given one or more variable declarations $x_0 \ldots x_n$ and a logic expression e, this states that the given variables declared in $x_0 \ldots x_n$ are universally quantified over e.
- **(exists** $x_0 \ldots x_n\, e$**):** Given one or more variable declarations $x_0 \ldots x_n$ and a logic expression e, this states that the given variables declared in $x_0 \ldots x_n$ are existentially quantified over e.
- **(var** x**):** This declares a variable x for use in a rule which is quantified either universally or existentially.
- **//** and **/* ... */:** Both expressions declare comments. The first comment variant is terminated by the end of line, whereas the second comment variant may span multiple lines.

In Table 5.1, a model-specific rule is shown which represents the rule used in the previous Example 5.2.14.

5.4 Applications of BSG Reasoning

The BSG Reasoning approach enables the formal description of a data format in a declarative manner by building on the BSG model defined in Chapter 4, which in turn is building on the formalisation given in Chapter 3. The *description of a PNG data format subset* is presented as an application in detail, followed a brief description of *format-aware fuzzing of bitstreams* in IT Security as an example for other potential applications:

```
(prefix bsg "http://bsg.org/bsgd/1.0/predicate/")
(prefix util "http://bsg.org/predicates/util/")
(prefix math "http://bsg.org/predicates/math/")

// M1: A bitstream segment with a source is a root segment
(forall (var segment) (var source) (var length)
  (implies
    (and
      (bsg:source segment source)
      (util:sourceLength source length)
    )
    (and
      (bsg:start segment 0)
      (bsg:length segment length)
    )
  )
)
```

Table 5.1: Example BSG rule which infers start and length of a root bitstream segment for which a source has been defined.

5.4.1 Description of a PNG data format subset

In order to describe a data format which exercises all elemental descriptive capabilities, a subset of the PNG raster image file format has been selected, where PNG images store compressed image data in multiple fragments through separate *IDAT chunks*, effectively a generalisation of the PNG raster image "litmus test" previously introduced in Chapter 3.4.

Selection of training and test corpora

A basis for sample bitstreams, the PNG Test Suite [vS98] includes 156 PNG images for compliance testing that exercise the format rules in sometimes extreme variants and also include three corrupted files. From this suite, both a *training corpus* and a *testing corpus* is selected:

- As a *training corpus*, a subset of images was selected that are structurally similar to the PNG raster image "litmus test", thereby exercising all four elementary descriptive capabilities. This subset consists of 8 files from the PNG Test Suite matching the filename pattern `OI??????.png`.
- As a *testing corpus*, all 153 valid PNG images were selected, excluding the three corrupted files. These also include PNG images with format-related properties which are not present in the training corpus, such as the use of transparency or images with indexed colour palettes.

For the training corpus, a *fitting set* of rules is to be constructed, where for every PNG image in the corpus, its deduced BSG instance shall have a coverage of 1.0.

Concerning the granularity of this fitting set of rules, primitive bitstream segments are allowed to represent arrays of encoded literals rather than segmenting these arrays into their elements. This decision is made in light of arrays of pixel data, where its segmentation into separate pixels and their individual decoding would lead to a bloated BSG instance description without a substantial descriptive benefit. Yet still, the fitting set of rules has to provide for the *structurally required granularity* that is necessary for parsing the bitstream completely.

Construction of a fitting set of rules

The fitting set of rules was constructed in an incremental fashion. Given a BSG instance with a coverage of less that 1.0, at least one generic bitstream segment necessarily exists in the graph. Through the addition of a rule, such a generic bitstream segment can then be deduced to be something more specific, thereby increasing the coverage of one or more BSG instances.

After reaching full coverage for the training corpus, excerpts of the resulting fitting set of rules are shown in Tables 5.2, 5.3 and 5.4. These three tables show 17 model-specific and the first 14 of 38 format-specific data format rules, with the entire set of rules listed in Appendix A:

- **Model-specific rules:** Model-specific rules, listed in Table 5.2, begin with rules on placement regarding a bitstream segment, starting 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 to ensure consistency (M5).

  Further rules include aspects of neighbourship of bitstream segments in a structure (M6 & M7), successorship of bitstream segments (M8-M12), placement in a structure (M13-M15) and resolvability (M16 & M17), which is necessary for decoding the contained literal of primitive bitstream segments.

- **Format-specific rules:** Format-specific rules for the selected PNG subset are listed in Tables 5.3 and 5.4. They start with a rule that deduces the PNG-specific type of 'png:root' for a bitstream source (F1). For such a bitstream segment, it can be deduced that there exists a first successor `?s` with `bsg:semantics(?s, 'png:signature')` (F2). For a 'png:signature', there exists a following PNG chunk as a 'png:chunk' structure (F3), as shown in Figure 5.1, which again always begins with a 'png:chunk-length' bitstream segment (F4), followed by a 'png:chunk-type' bitstream segment (F5).

  The actual composition of a PNG chunk may vary. 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

#	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 5.2: List of model-specific rules (M1-17) [HBSM08].

#	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('F4', ?c, ?l)` ∧ `bsg:firstSuccessor(?c, ?l)` ∧ `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')`

Table 5.3: Excerpt of 36 format-specific rules for a subset of the PNG raster image file format, listing rules F1-F7 [HBSM08].

#	Rule
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(?l, 'png:chunk-length')` → `bsg:type(?l, 'bsg:primitive')` ∧ `bsg:encoding(?t, 'http://www.dataformats.net/2008/04/bsg-encodings#msbf-uint')` ∧ `bsg:length(?l, 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(?cr, 'png:chunk-crc')` → `bsg:type(?t, 'bsg:primitive')` ∧ `bsg:encoding(?t, 'http://www.dataformats.net/2008/04/bsg-encodings#msbf-uint')` ∧ `bsg:length(?cr, 32)`
F12	`bsg:semantics(?ch, 'png:chunk')` ∧ `bsg:semantics(?t, 'png:chunk-type')` ∧ `bsg:successor(?ch, ?t)` ∧ `bsg:value(?t, ?v)` → `util:concat('png:chunk:', ?v, ?ct)` ∧ `bsg:semantics(?ch, ?ct)`
F13	`bsg:semantics(?c, 'png:chunk')` ∧ `bsg:end(?c, ?ce)` ∧ `bsg:successor(?r, ?c)` ∧ `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(?c, 'png:chunk')` ∧ `bsg:end(?c, ?ce)` ∧ `bsg:successor(?r, ?c)` ∧ `bsg:length(?r, ?rl)` ∧ `math:eq(?ce, ?rl)` → `bsg:lastSuccessor(?r, ?c)`

Table 5.4: Excerpt of 36 format-specific rules for a subset of the PNG raster image file format, listing rules F8-F14 [HBSM08].

deduced from the 'png:chunk-type' value and assigned as `bsg:semantics` to the chunk (F12). The remaining rules listed in Table 5.4 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 aspects depending on the specific chunk type, such as for the IHDR chunk which contains information on image width and height.

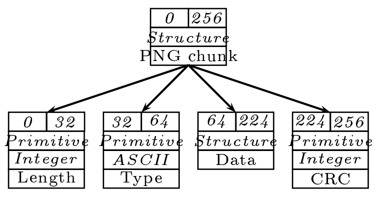


Figure 5.1: BSG instance for a PNG chunk.

Applying these rules to an actual PNG image for the deduction of a BSG instance can be followed exemplary in the following two deduction steps. Intending to deduce a BSG instance for the image `OI2N0G16.PNG`, the following initial fact is given:

```
bsg:source('root','oi2n0g16.png')                    (5.1)
```

The inference process tries to apply all rules to obtain new facts. In the first step, only the rules F1 and M1 are applicable, which yield the following new facts in addition:

```
bsg:semantics('root','png:root')∧
bsg:start('root',0)∧
bsg:length('root',1432)                              (5.2)
```

Again, the inference 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 new facts in addition:

```
bsg:type('root','bsg:structure')∧
bsg:firstSuccessor('root','_sc1')∧
bsg:semantics('_sc1','png:signature')∧
bsg:end('root',1432)                                 (5.3)
```

The process of inference is repeated until no new facts can be inferred. The resulting facts from the reached fixed point describe a BSG instance for the PNG image oi2n0g16.png, which is part of the training corpus and thus has complete coverage.

Exclusion of corrupted PNG images

The fitting set of rules for the training corpus were evaluated on the testing corpus consisting of all 153 valid PNG images from the PNG Test Suite, specifically excluding three corrupt images from the suite. These three corrupt images were excluded from the evaluation, since the fitting set of rules does not include verifying rules for PNG-specific properties. Verifying rules for PNG-specific properties cause a contradiction in the conclusion when such a PNG-specific property is violated by a corrupt PNG raster image. This is very similar to the model-specific rule M5 in Table 5.2, which verifies the model-specific property of BSG that the start, length and end properties of a bitstream segment add up. Since this description of the PNG data format subset is focused on a practical exercise of all four elementary descriptive capabilities in a real-life use case, verifying rules in the fitting set are limited to model-specific properties of BSG, but may be extended through future work.

Evaluation of the fitting set of rules

Evaluating the resulting coverage of deduced BSG instance on the testing corpus, both the *original fitting set of rules* and an *extended fitting set of rules* were evaluated:

- **Original fitting set of rules:** Data format knowledge contained in the fitting set of rules was sufficiently complete as to describe the composition of 64 of 153 PNG images completely, where every deduced BSG instance has a coverage of 1.0. For the remaining 89 PNG images, the fitting set of rules is incomplete, as their deduced BSG instances have an average coverage of 0.79. Coverage and the number of iterative steps required are listed in Appendix B for each file in the testing set.

  This can be explained by the existence of specialised data structures that have not been present in the training corpus for which the fitting rules were defined. Still, even in the absence of complete coverage of such data, an average coverage of 0.79 signifies that essential aspects of the PNG data format have been captured in the fitting set of rules as it is. Constructing a fitting set of rules for the test corpus and increasing the average coverage by 0.21 requires additional rules for handling colour palette information (PLTE and sPLT chunks), transparency information (tRNS chunk), background colour (bKGD chunk), textual data (tEXt and zTXt chunks) and other types of data with increasingly rare occurrences.

- **Extended fitting set of rules:** To estimate the effect of adding further rules, two preliminary rules, F37 and F38 listed in Appendix A, were added to form an extended fitting set of rules. These rules served for handling palette information stored in PLTE chunks, and evaluated the extended fitting rules on all 153 PNG images as well. Complete coverage was then achieved for 78 PNG images, while the remaining 75 PNG images have an average coverage of 0.91. Adding these two rules increased the number of BSG instances with full coverage by 14, while the average coverage on the remaining BSG instances increased by 0.12.

  Further step-by-step increments can be achieved by constantly adding rules for handling increasingly rare types of data. Just as during the construction, this can be achieved in each step by searching incomplete BSG instances for generic bitstream segments that “cause” the incompleteness, by matching these bitstream segments with existing data format knowledge in the natural-language PNG specification, and by introducing a rule which explains these generic bitstream segments a bit further. Extending the fitting set of rules for increased coverage therefore introduces no changes to the presented BSG model or the BSG Reasoning model itself, and is thus left for future work.

In the evaluation of both the original and the extended fitting set of rules, the deduction process computed a fixed point and halted on all instances. As previously examined in the analysis in Chapter 3, termination cannot be guaranteed when an approach is sufficiently expressive to describe arbitrary data formats. This is reflected in the BSG Reasoning approach as well, since errors may be present in an erroneous set of rules preventing a fixed point to be reached. During the construction of fitting sets of rules and their evaluation, the issue of potentially non-terminating, erronous sets of rules was addressed through a simple heuristic, where a limit on the number of deduction steps is placed, aborting the deduction process beyond that limit.

Regarding the number of iterative steps necessary, it was observed that the typical number of iterative steps required for the fitting ruleset to reach a fixed point on valid PNG images ranges from 72 up to 170 steps, with two structurally exceptional PNG images from the PNG Test Suite requiring 1389 and 3279 iterative steps, respectively. In both oi9n0g16.png and oi9n2c16.png files, which were also present in the training corpus, compressed and transformed image data is fragmented into bitstream segments with 8 bit length each, each and every byte encapsuled into a separate IDAT chunk. The generic PNG chunk data structure is of variable length, and since there is only one length descriptor at its start, the placement of chunks has to be resolved from the beginning of the bitstream to the end. This extreme case of fragmentation leads to a substantial increase in required iterative steps, which is inherent in the PNG data format and therefore is to affect any other declarative approach or procedural implementation as well. Both files can be considered an extreme example, but demonstrate what is still considered legal in terms of the PNG data format 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-naïve evaluation method for the deduction process as known from Datalog [Ull89] is absolutely essential.

5.4.2 Format-aware fuzzing of bitstreams

This application of BSG Reasoning is given in brief overview. Typical fuzzing of bitstreams in IT Security tests applications for their robustness by introducing bit errors into files and passes the manipulated files to applications. If applications exhibit erroneous behaviour or crash, then this is an indication of robustness-related issues in an application, which may be exploitable.

Yet, standard fuzzing is format-unaware, as introducing bit errors does not take into account where the bit error is actually introduced, reducing the efficiency of the fuzzing process. From an application's point of view, bit errors introduced through fuzzing are virtually the same as bit errors introduced through a transport, and applications may take measures such as cyclic redundancy codes (CRCs) or even error-correcting codes to counter these. Format-unaware fuzzing of data formats employing such countermeasures is therefore inefficient, as it is extremely unlikely that random bit errors will simultaneously introduce an critical bit error in data and change the dependent CRC or error-correcting codes accordingly.

Data format knowledge represented through BSG Reasoning rulesets provides a basis for a more efficient, format-aware fuzzing approach. When changing the value of a bitstream segment, rule-induced dependencies such as a CRC code spanning a number of bitstream segments can be detected and used for "repairing" dependent values recursively. This way, an application cannot detect the bit errors introduced by fuzzing directly, and previously protected portions of code can be subjected to erroneous data.

Using the BSG model and BSG Reasoning approach for format-aware fuzzing provides further benefits. The approach detailed above also allows for more sophisticated modifications such as resizing variable-length blocks of data, or specifically focusing on bitstream segments that are packaging, and thus potentially relevant for processing, instead of targetting payload which may have no direct effects on processing.

5.5 Discussion

As with the BSG model, the presented BSG Reasoning approach has several distinct features over examined related work, such as providing a *formal description in a declarative manner with universal applicability*, which leads to its *applicability for different format-related use cases.* The approach also allows the *measurable, incremental construction of rulesets* and enables reuse through *modular descriptions.*

- **Formal description in a declarative manner with universal applicability:** In contrast to the examined approaches from related work such as XCEL, the BSG Reasoning approach provides support for all four elementary descriptive capabilities down to bit granularity by building on the BSG model. Data format knowledge is formally described as logic rules in a declarative manner.

- **Applicability for different format-related use cases:** The specification of data format knowledge in a declarative manner rather than through procedural descriptions enables the use of data format rules in unforeseen contexts. While the need for reading and writing format-compliant data often triggers the specification of data format knowledge in the first place, other concerns depend on the same knowledge, yet have different constraints and process format-compliant data in a different manner, as previously shown with the second outlined application of BSG Reasoning, the format-aware fuzzing of bitstreams in IT Security. In contrast to declarative descriptions, procedural descriptions of data formats through approaches such as Flavor cannot be applied directly to such varying use cases.

- **Measurable, incremental construction of rulesets:** The BSG Reasoning approach supports the incremental construction of rulesets for data formats. A given ruleset can be tested against a set of format-compliant bitstreams by evaluating the ruleset against every bitstream and computing the coverage of the resulting BSG instance, averaging the coverage values of all instances. For a ruleset with an overall coverage of less than 1, a BSG instance exists where there is a generic segment present. In order to increase the coverage of such an instance, one or more rules addressing the given generic segment can be defined and added to the ruleset. This allows for a "divide-and-conquer" approach in the construction of fitting sets of rules, such as for the documentation of legacy data formats in Digital Preservation, or when a new, customised data format needs to be constructed.

- **Modular description:** It is desirable to specify data format rules separately, allowing to incrementally describe a data format along existing instances and examples. Rather than computing a bitstream segment graph using a reversible Turing machine in some opaque manner, it is preferable to split the transformation into separate Turing machines that can be reused, and describe the data format using an incremental set of rules, building on a uniform model for data format instances.

Besides these features of the BSG Reasoning approach, two aspects were observed that are worth pointing out, namely the *substitution of functions with relations and variables* and the *redundancy of information* typically present in data formats:

- **Substitution of functions with relations and variables:** In contrast to other logic systems such as Prolog, the BSG reasoning system does not support functions, but substitutes them with the use of relations and free variables. Assuming a predicate $math : eq/2$ over the set of all equal arguments and a function $math : plus/2$ adding both arguments and returning the result, an expression like $math : eq(math : plus(2, 2), 4)$ is not supported. Instead, it can be expressed as $math : sum(2, 2, ?a) \vee math : eq(?a, 4)$ introducing the variable $?a$. This design decision provides a unified approach to designing and implementing both functions and relations in the reasoning system. The design decision therefore simplifies the implementation of the reasoning system.

- **Redundancy of information:** It is notable that there often is a redundancy of information related to data format knowledge. As previously observed, specific information concerning the composition of data can often be inferred in more than one way.

  Example 5.5.1: Let d be a data format. Every format-compliant bitstream $x \in \mathbb{B}_d$ is a structure composed from one or more blocks. Again, every block is a structure which consists of a tag field, a length field and a data field, where both the tag and length field have a fixed, known size and encoding, and the length field contains the length of the entire block. Given $y \in \mathbb{B}_d$ be a format-compliant bitstream with an overall length l, where there is a last block z with a known start position. Knowing that z is the last block, the information contained in the length field of z is redundant, as the length can also be inferred from the start position of z and the overall length l.

  When processing format-compliant information, the redundancy of information inherent in a data format requires consistency to be maintained when writing as well as checking for consistency when reading. This has implications for IT Security, where missing consistency checks in applications can result in a vulnerability for buffer overflow attacks. In a *buffer overflow attack*, an application is tricked into copying data into a buffer which is too small, thus overwriting and replacing adjacent machine code of the application with malicious code embedded into the data by an attacker.

5.6 Summary

This chapter defines the BSG Reasoning approach as a formal, declarative aproach for data format description with universal applicability which extends the BSG model previously presented in Chapter 4.

For the BSG Reasoning approach, a formal language is defined where data format rules can be specified using predicates expressing propositions about the composition of data based on the BSG model. In an inference process operating on a given bitstream, these rules can be matched to a knowledge base of facts in a semi-naïve fashion, allowing further facts to be inferred until a least fixed-point is reached, which describes the corresponding BSG instance for the given bitstream. In order to exchange and store these rules, a compact representation of rulesets is introduced which has been inspired by XML Common Logic.

Demonstrating its capabilities, the application of BSG Reasoning is shown to full coverage of a subset of the PNG raster image file format which corresponds to the PNG image "litmus test" from Chapter 3, exercising all elementary descriptive capabilities. When evaluating this fitting set of 53 data format rules on a larger corpus of valid PNG images, 64 of 153 samples were described to full coverage, with the remaining 89 samples having an average coverage of 0.79. When extending the fitting set of rules with two rules for palette colour information, 78 of 153 samples were described to full coverage, with the remaining 75 samples having an average coverage of 0.91. The termination of the reasoning process has to be guarded using heuristics, which is a common problem to all approaches for describing arbitrary data formats, as observed in Chapter 3. Another application of BSG Reasoning is outlined with the format-aware fuzzing of bitstreams in IT Security.

In the following discussion, central features of the BSG Reasoning approach are illuminated which distinguish it from examined related work, such as its universal applicability or its support of measurable, incremental construction of rulesets.

Chapter 6

Finale

6.1 Introduction

This finale concludes the thesis with a *retrospection*, some *conclusions* and an *outlook* regarding future directions of format-related research in combination with the BSG model and the BSG Reasoning approach contributed by this thesis.

6.2 Retrospection

In retrospect, a number of contributions have been made in the course of this thesis:

- Chapter 2 surveyed a number of approaches in literature which have the goal of describing data formats. Some of these approaches claim their suitability for describing arbitrary data formats, yet no proof is given. This resulted in the question of how to formalise the concept of a data format, and what properties of a formalisation are required for general applicability.

- Chapter 3 provided a research hypothesis for improving data format description, which is used to formalise the concept of data formats and for further analysis. The analysis showed inherent properties of data formats and presented resulting limitations for modelling arbitrary data formats. Moreover, the analysis provided the concept of elemental descriptive capabilities which are required for describing arbitrary data formats, and presented a PNG raster image as “lithmus test” which requires support for all elemental descriptive capabilities. Assessing and comparing the surveyed approaches in literature shows that contrary to some claims, they cannot describe arbitrary data formats, as they lack the required descriptive capabilities. An exception to that observation was XCEL, which is almost complete in terms of its descriptive capablities.

- Chapter 4 provided a model for describing a single instance of a data format through the Bitstream Segment Graph (BSG) model, which employs the required descriptive capabilities previously identified. The chapter formalised the BSG model, presented a set of operations under which arbitrary BSG instances can be produced, and gave both visual and RDF/N3-based digital representations. The BSG model was finally used for describing the PNG raster image “litmus test” and for describing the composition of an exploit in IT Security.

- Chapter 5 built on the BSG model and provided an approach for describing potentially infinite sets of data format instances through the BSG Reasoning approach. The presented approach uses logic rules in order to infer a BSG instance over a given bitstream. The BSG Reasoning approach was finally applied in the description of a subset of the PNG raster image file format that includes the "lithmus test", reaching full coverage on a test corpus and near-complete coverage on an extended test corpus.

6.3 Conclusions

Describing data formats is a task that is inherently complex. From understanding legacy data formats that have not been publicly disclosed to data formats for which documentation has been lost, it is about finding out which bitstream segments serve which purpose, either serving as packaging that provides information on the composition itself, or as payload which carries part of the actual information to be exchanged.

The BSG model and the BSG Reasoning approach have a solid foundation in theoretical terms, yet in practice they depend on the availability of fitting coding and transformation functions, as well as suitable predicates for the inference process. Yet, even with the relatively small number of coding and transformation function as well as predicates defined in this thesis, it was possible to describe a non-trivial subset of the PNG raster image format.

The complexity of describing data formats is a task that depends on the inherent complexity of the format as required to fulfill its function, and is increased by additional "protection considerations" or "legacy support" that its creators may have had in mind.

6.4 Outlook

There are a number of potential opportunities for future research related to the BSG model and the BSG Reasoning approach. Here, the *computer-aided reverse-engineering of data format rules*, the *use of reversible programming languages for coding and transformation functions*, the *analysis of space-efficiency regarding existing data formats*, and *format-aware fuzzing of bitstreams* are given as examples.

6.4.1 Computer-aided reverse-engineering of data format rules

Reverse-engineering of legacy data formats and their governing rules is a non-trivial task due to the potential complexity of data format rules and the sheer enormous search space of potentially valid data format rules.

Regarding a small portion of a bitstream, experienced human engineers may be capable of making sense of patterns they observe, for example by recognising bit patterns with defined meanings, such as start codes in MPEG-2 Transport Streams. Using the BSG model, the composition of such bitstreams can be documented manually, and using the BSG Reasoning approach, fitting sets of rules can be constructed manually and evaluated regarding their coverage.

In order to generalise complex data format rules that were observed, such a process has to be repeated over a representative amount of samples. For complex data formats without sufficient documentation or easily apparent similarities to existing data formats, generalising potential data format rules and testing them over the entire corpus of samples is not feasible for human engineers alone, since the actual process of defining rulesets, considering alternatives and testing these is still unaided in terms of *Human Computer Interaction (HCI)* considerations.

Therefore, *computer-aided reverse-engineering* of data format rules may have some potential to improve reverse-engineering efforts for legacy data formats. When a human engineer may identify and document the composition of specific data, potentially valid rules may be generated and tested automatically against the entire corpus. Rather than representing data format rules as text, future work on a graphical representation of data format rules may aid human engineers in their understanding, addressing HCI concerns. Fitting rule examples may then be proposed in such a manner to the engineer, who may decide to include the rule and to apply it to the entire corpus, focusing on potential problems and remaining generic bitstream segments in individual BSG instances.

6.4.2 Use of reversible programming languages

Implementing coding and transformation functions that ensure the preservation of information is difficult, as design mistakes or programming errors have to be prevented.

While in theory, it is possible to ensure bijectivity of such functions and thus the lossless mapping of information between different representations through the use of Reversible Turing Machines, reversible programming languages are nowhere near standard, established programming languages such as Java or C/C++ regarding tool support or best practices for developers.

It would be highly interesting to allow the implementation of coding and transformation functions through reversible programming languages that would guarantee information preservation during mappings, and thus to ensure the mapping between internal and external representations to be correct. For example, the implementation of an established lossless compression algorithm in a reversible programming language could be a complex, but interesting goal in terms of future research.

6.4.3 Analysis of space-efficiency regarding existing data formats

The design of data formats depends on a multitude of factors, such as catering for space efficiency. During the design process, a number of assumptions are made, for example concerning the statistic distribution of primitive values which may favour a certain type of encoding as it is considered to be more space-efficient.

By using the BSG model on a representative sample of format-compliant bitstreams for a certain data format, it is possible to analyse the statistical distribution of a certain type of primitive values, which would allow statements to be made on the space-efficiency of design choices in existing data formats, and to extract knowledge that may improve future design choices.

Appendix A

BSG Reasoning ruleset for PNG subset

```
(prefix bsg "http://bsg.org/bsgd/1.0/predicate/")
(prefix util "http://bsg.org/predicates/util/")
(prefix math "http://bsg.org/predicates/math/")

// M1
(forall
  (var segment) (var source) (var length)
  (implies
    (and
      (bsg:source segment source)
      (util:sourceLength source length)
    )
    (and
      (bsg:start segment 0)
      (bsg:length segment length)
    )
  )
)

// M2
(forall
  (var segment) (var start) (var length) (var end)
  (implies
    (and
      (bsg:length segment length)
      (bsg:end segment end)
      (math:sum start length end)
    )
    (bsg:start segment start)
  )
)

// M3
(forall
  (var segment) (var start) (var length) (var end)
  (implies
    (and
      (bsg:start segment start)
      (bsg:end segment end)
      (math:sum start length end)
    )
    (bsg:length segment length)
  )
)

// M4
(forall
  (var segment) (var start) (var length) (var end)
  (implies
    (and
      (bsg:start segment start)
      (bsg:length segment length)
      (math:sum start length end)
    )
    (bsg:end segment end)
  )
)

// M5
(forall
  (var segment) (var start) (var length) (var end)
  (implies
    (and
      (bsg:start segment start)
      (bsg:length segment length)
      (bsg:end segment end)
    )
    (math:sum start length end)
  )
)

// M6
(forall
  (var leader) (var follower)
  (iff
    (bsg:leads leader follower)
    (bsg:follows follower leader)
  )
)

// M7
(forall
  (var previous) (var next) (var position)
  (iff
    (and
      (bsg:leads previous next)
      (bsg:end previous position)
    )
    (and
      (bsg:start next position)
      (bsg:follows next previous)
    )
  )
)

// M8
(forall
  (var predecessor) (var successor)
  (implies
    (bsg:firstSuccessor predecessor successor)
    (bsg:successor predecessor successor)
  )
)

// M9
(forall
  (var predecessor) (var successor)
  (implies
    (bsg:lastSuccessor predecessor successor)
    (bsg:successor predecessor successor)
  )
)

// M10
(forall
  (var predecessor) (var successor)
  (iff
    (bsg:successor predecessor successor)
    (bsg:predecessor successor predecessor)
  )
)

// M11
(forall
  (var predecessor) (var previousSuccessor)
  (var nextSuccessor)
  (implies
    (and
      (bsg:successor predecessor previousSuccessor)
      (bsg:leads previousSuccessor nextSuccessor)
    )
    (bsg:successor predecessor nextSuccessor)
  )
)
```

```

// M12
(forall
  (var predecessor) (var previousSuccessor)
  (var nextSuccessor)
  (implies
    (and
      (bsg:successor predecessor nextSuccessor)
      (bsg:follows nextSuccessor previousSuccessor)
    )
    (bsg:successor predecessor previousSuccessor)
  )
)

// M13
(forall
  (var predecessor) (var successor)
  (implies
    (bsg:firstSuccessor predecessor successor)
    (bsg:start successor 0)
  )
)

// M14
(forall
  (var predecessor) (var successor) (var length)
  (implies
    (and
      (bsg:length predecessor length)
      (bsg:lastSuccessor predecessor successor)
    )
    (and
      (bsg:end successor length)
    )
  )
)

// M15
(forall
  (var predecessor) (var successor) (var end)
  (implies
    (and
      (bsg:end successor end)
      (bsg:lastSuccessor predecessor successor)
    )
    (bsg:length predecessor end)
  )
)

// M16
```

```
(forall
  (var root) (var rootStart) (var rootLength) (var rootEnd)
  (var rootType) (var rootSource)
  (implies
    (and
      (bsg:start root rootStart)
      (bsg:length root rootLength)
      (bsg:end root rootEnd)
      (bsg:type root rootType)
      (bsg:source root rootSource)
    )
    (bsg:resolved root true)
  )
)

// M17
(forall
  (var segment) (var segStart)
  (var segType) (var segPredecessor)
  (implies
    (and
      (bsg:start segment segStart)
      (bsg:type segment segType)
      (bsg:successor segPredecessor segment)
      (bsg:resolved segPredecessor true)
    )
    (bsg:resolved segment true)
  )
)

// F1
(forall
  (var root)
  (var path)

  (implies
    (bsg:source root path)
    (bsg:semantics root "png:root")
  )
)

// F2
(forall
  (var root)
  (exists
    (var signature)
    (implies
      (bsg:semantics root "png:root")
      (and
        (bsg:firstSuccessor root signature)
        (bsg:semantics signature "png:signature")
      )
    )
  )
)

// F3
(forall
  (var signature)
  (exists
    (var firstChunk)
    (implies
      (bsg:semantics signature "png:signature")
      (and
        (bsg:leads signature firstChunk)
        (bsg:semantics firstChunk "png:chunk")
      )
    )
  )
)

// F4
(forall
  (var chunk) (var chunkEnd) (var root) (var fileLength)
  (exists
    (var nextChunk)
    (implies
      (and
        (bsg:semantics chunk "png:chunk")
        (bsg:end chunk chunkEnd)
        (bsg:predecessor chunk root)
        (bsg:length root fileLength)
        (math:lt chunkEnd fileLength)
      )
      (and
        (bsg:leads chunk nextChunk)
        (bsg:semantics nextChunk "png:chunk")
      )
    )
  )
)

// F5
(forall
  (var chunk) (var chunkEnd) (var root) (var rootLength)
  (implies
    (and
      (bsg:semantics chunk "png:chunk")
      (bsg:end chunk chunkEnd)
      (bsg:successor root chunk)
      (bsg:length root rootLength)
      (math:eq chunkEnd rootLength)
    )
    (and
      (bsg:lastSuccessor root chunk)
    )
  )
)

// F6
(forall
  (var chunk)
  (exists
    (var length)
    (implies
      (and
        (bsg:semantics chunk "png:chunk")
      )
      (and
        (bsg:firstSuccessor chunk length)
        (bsg:semantics length "png:chunk-length")
        (bsg:length length 32)
      )
    )
  )
)

// F7
(forall (var length)
  (exists (var type)
    (implies
      (and
        (bsg:semantics length "png:chunk-length")
      )
      (and
        (bsg:leads length type)
        (bsg:semantics type "png:chunk-type")
        (bsg:length type 32)
      )
    )
  )
)

// F8
(forall (var length) (var type) (var chunk)
  (exists (var crc)
    (implies
      (and
        (bsg:semantics length "png:chunk-length")
        (bsg:value length 0)
        (bsg:leads length type)
        (bsg:predecessor length chunk)
      )
      (and
        (bsg:lastSuccessor chunk crc)
        (bsg:follows crc type)
        (bsg:semantics crc "png:chunk-crc")
      )
    )
  )
)

// F9
(forall
  (var crc) (var type) (var crcValue)
  (implies
    (and
      (bsg:semantics type "png:chunk-type")
      (bsg:leads type crc)
      (bsg:semantics crc "png:chunk-crc")
      (bsg:resolved type true)
      (util:crc type crcValue)
    )
    (png:crc crc crcValue)
  )
)

// F10
(forall
  (var length) (var lengthByteValue)
  (var lengthBitValue) (var type)
  (exists
    (var data)
    (var crc)

    (implies
      (and
        (bsg:semantics length "png:chunk-length")
        (bsg:value length lengthByteValue)
        (math:lt 0 lengthByteValue)
        (bsg:leads length type)
        (bsg:predecessor length chunk)
        (math:product lengthByteValue 8 lengthBitValue)
      )
      (and
        (bsg:lastSuccessor chunk crc)
        (bsg:leads type data)
        (bsg:leads data crc)
        (bsg:length data lengthBitValue)
        (bsg:semantics data "png:chunk-data")
        (bsg:semantics crc "png:chunk-crc")
      )
    )
  )
)

// F11
(forall
  (var length)
  (var byteLength)
  (var bitLength)
  (var type)
  (var data)

  (implies
    (and
      (bsg:semantics length "png:chunk-length")
      (bsg:leads length type)
      (bsg:leads type data)
      (bsg:semantics data "png:chunk-data")
      (bsg:value length byteLength)
      (math:product byteLength 8 bitLength)
    )
    (bsg:length data bitLength)
  )
)

// F12
(forall
  (var crc)
  (implies
    (bsg:semantics crc "png:chunk-crc")
    (bsg:length crc 32)
  )
)

// F13
(forall
  (var chunkSegment) (var typeSegment) (var typeValue)
  (var chunkType)
  (implies
    (and
      (bsg:semantics chunkSegment "png:chunk")
      (bsg:semantics typeSegment "png:chunk-type")
      (bsg:successor chunkSegment typeSegment)
      (bsg:value typeSegment typeValue)
    )
    (and
      (util:concat "png:chunk:" typeValue chunkType)
      (bsg:semantics chunkSegment chunkType)
    )
  )
)

// F14
(forall
  (var signature)
  (exists
    (var a) (var b) (var c) (var d) (var e) (var f)
    (var g) (var h)
    (implies
      (bsg:semantics signature "png:signature")
      (and
        (bsg:firstSuccessor signature a)
        (bsg:leads a b)
        (bsg:leads b c)
        (bsg:leads c d)
        (bsg:leads d e)
        (bsg:leads e f)
        (bsg:leads f g)
        (bsg:leads g h)
        (bsg:lastSuccessor signature h)
      )
    )
  )
)

// F15
(forall
  (var signature) (var sig_char)
  (implies
    (and
      (bsg:semantics signature "png:signature")
      (bsg:successor signature sig_char)
    )
    (and
      (bsg:type sig_char "bsg:primitive")
      (bsg:encoding sig_char
        "http://www.dataformats.net/2009/01/25-bsg-ext-ns
          #encoder:msbf-uint")
      (bsg:length sig_char 8)
      (bsg:semantics sig_char "png:sig_char")
    )
  )
)

// F16
(forall
  (var segment)
  (implies
    (or
      (bsg:semantics segment "png:root")
      (bsg:semantics segment "png:signature")
      (bsg:semantics segment "png:chunk")
    )
    (bsg:type segment "bsg:structure")
  )
)

// F17
(forall
  (var segment) (var value) (var encoding)
  (implies
    (and
      (bsg:type segment "bsg:primitive")
      (bsg:encoding segment encoding)
      (bsg:resolved segment true)
      (util:value segment value)
    )
    (bsg:value segment value)
  )
)

// F18
(forall
  (var chunkLength)
  (implies
    (bsg:semantics chunkLength "png:chunk-length")
    (and
      (bsg:type chunkLength "bsg:primitive")
      (bsg:encoding chunkLength
        "http://www.dataformats.net/2009/01/25-bsg-ext-ns
          #encoder:msbf-uint")
    )
  )
)

// F19
(forall
  (var chunkType)
  (implies
    (bsg:semantics chunkType "png:chunk-type")
    (and
      (bsg:type chunkType "bsg:primitive")
      (bsg:encoding chunkType
        "http://www.dataformats.net/2009/01/25-bsg-ext-ns
          #encoder:ascii-string")
    )
  )
)
```

```

// F20
(forall
  (var chunkCrc)
  (implies
    (bsg:semantics chunkCrc "png:chunk-crc")
    (and
      (bsg:type chunkCrc "bsg:primitive")
      (bsg:encoding chunkCrc
        "http://www.dataformats.net/2009/01/25-bsg-ext-ns
          #encoder:msbf-uint")
    )
  )
)

// F21
(forall
  (var chunkSegment) (var dataSegment)
  (implies
    (and
      (bsg:semantics chunkSegment "png:chunk:gAMA")
      (bsg:semantics dataSegment "png:chunk-data")
      (bsg:successor chunkSegment dataSegment)
    )
    (and
      (bsg:type dataSegment "bsg:primitive")
      (bsg:semantics dataSegment "png:gamma-value")
      (bsg:encoding dataSegment
        "http://www.dataformats.net/2009/01/25-bsg-ext-ns
          #encoder:msbf-uint")
    )
  )
)

// F22
(forall
  (var chunkSegment) (var dataSegment)
  (exists
    (var widthSegment) (var heightSegment)
    (var bitDepthSegment) (var colorTypeSegment)
    (var compressionMethodSegment) (var filterMethodSegment)
    (var interlaceMethodSegment)
    (implies
      (and
        (bsg:semantics chunkSegment "png:chunk:IHDR")
        (bsg:semantics dataSegment "png:chunk-data")
        (bsg:successor chunkSegment dataSegment)
      )
      (and
        (bsg:type dataSegment "bsg:structure")
        (bsg:firstSuccessor dataSegment widthSegment)
        (bsg:leads widthSegment heightSegment)
        (bsg:leads heightSegment bitDepthSegment)
        (bsg:leads bitDepthSegment colorTypeSegment)
        (bsg:leads colorTypeSegment compressionMethodSegment)
        (bsg:leads compressionMethodSegment
          filterMethodSegment)
        (bsg:leads filterMethodSegment interlaceMethodSegment)
        (bsg:lastSuccessor dataSegment interlaceMethodSegment)

        (bsg:semantics widthSegment "png:width")
        (bsg:length widthSegment 32)
        (bsg:type widthSegment "bsg:primitive")
        (bsg:encoding widthSegment
          "http://www.dataformats.net/2009/01/25-bsg-ext-ns
            #encoder:msbf-uint")

        (bsg:semantics heightSegment "png:height")
        (bsg:length heightSegment 32)
        (bsg:type heightSegment "bsg:primitive")
        (bsg:encoding heightSegment
          "http://www.dataformats.net/2009/01/25-bsg-ext-ns
            #encoder:msbf-uint")

        (bsg:semantics bitDepthSegment "png:bitDepth")
        (bsg:length bitDepthSegment 8)
        (bsg:type bitDepthSegment "bsg:primitive")
        (bsg:encoding bitDepthSegment
          "http://www.dataformats.net/2009/01/25-bsg-ext-ns
            #encoder:msbf-uint")

        (bsg:semantics colorTypeSegment "png:colorType")
        (bsg:length colorTypeSegment 8)
        (bsg:type colorTypeSegment "bsg:primitive")
        (bsg:encoding colorTypeSegment
          "http://www.dataformats.net/2009/01/25-bsg-ext-ns
            #encoder:msbf-uint")

        (bsg:semantics compressionMethodSegment
          "png:compressionMethod")
        (bsg:length compressionMethodSegment 8)
        (bsg:type compressionMethodSegment "bsg:primitive")
        (bsg:encoding compressionMethodSegment
          "http://www.dataformats.net/2009/01/25-bsg-ext-ns
            #encoder:msbf-uint")

        (bsg:semantics filterMethodSegment "png:filterMethod")
        (bsg:length filterMethodSegment 8)
        (bsg:type filterMethodSegment "bsg:primitive")
        (bsg:encoding filterMethodSegment
          "http://www.dataformats.net/2009/01/25-bsg-ext-ns
            #encoder:msbf-uint")

        (bsg:semantics interlaceMethodSegment
          "png:interlaceMethod")
        (bsg:length interlaceMethodSegment 8)
        (bsg:type interlaceMethodSegment "bsg:primitive")
        (bsg:encoding interlaceMethodSegment
          "http://www.dataformats.net/2009/01/25-bsg-ext-ns
            #encoder:msbf-uint")
      )
    )
  )
)

// F23
(forall
  (var predecessor)
  (var firstSuccessor)
  (implies
    (bsg:firstSuccessor predecessor firstSuccessor)
    (bsg:leftAnchored firstSuccessor)
  )
)

// F24
(forall
  (var predecessor)
  (var lastSuccessor)
  (implies
    (bsg:lastSuccessor predecessor lastSuccessor)
    (bsg:rightAnchored lastSuccessor)
  )
)

// F25
(forall
  (var previous)
  (var next)
  (implies
    (and
      (bsg:leads previous next)
      (bsg:leftAnchored previous)
    )
    (bsg:leftAnchored next)
  )
)

// F26
(forall
  (var previous)
  (var next)

  (implies
    (and
      (bsg:leads previous next)
      (bsg:rightAnchored next)
    )
    (bsg:rightAnchored previous)
  )
)


// F27
(forall
  (var segment)

  (implies
    (and
      (bsg:leftAnchored segment)
      (bsg:rightAnchored segment)
    )
    (bsg:anchored segment)
  )
)


// Number all chunks depending on IDAT type
// F28
(forall
  (var rootSegment)
  (var firstChunkSegment)

  (implies
    (and
      (bsg:semantics rootSegment "png:root")
      (bsg:firstSuccessor rootSegment firstChunkSegment)
      (bsg:anchored firstChunkSegment)
    )
    (tmp:order firstChunkSegment 0)
  )
)

// F29
(forall
  (var previousChunkSegment)
  (var nextChunkSegment)
  (var previousIndex)
  (var nextIndex)
  (var nextChunkSemantics)

  (implies
    (and
      (tmp:order previousChunkSegment previousIndex)
      (bsg:leads previousChunkSegment nextChunkSegment)
      (bsg:anchored nextChunkSegment)
      (not
        (bsg:semantics nextChunkSegment "png:chunk:IDAT")
      )
    )
    (tmp:order nextChunkSegment previousIndex)
  )
)

// F30
(forall
  (var previousChunkSegment)
  (var nextChunkSegment)
  (var previousIndex)
  (var nextIndex)
  (var nextChunkSemantics)

  (implies
    (and
      (tmp:order previousChunkSegment previousIndex)
      (bsg:leads previousChunkSegment nextChunkSegment)
      (bsg:anchored nextChunkSegment)
      (bsg:semantics nextChunkSegment "png:chunk:IDAT")
    )
    (and
      (math:sum previousIndex 1 nextIndex)
      (tmp:order nextChunkSegment nextIndex)
    )
  )
)

// Assign png:compressed to an individual png:chunk:IDAT
// F31
(forall
  (var rootSegment)
  (var chunkSegment)
  (var lastChunkSegment)
  (var dataSegment)
  (var index)

  (implies
    (and
      (bsg:lastSuccessor rootSegment lastChunkSegment)
      (bsg:semantics rootSegment "png:root")
      (bsg:semantics lastChunkSegment "png:chunk")
      (tmp:order lastChunkSegment 1)

      (bsg:successor rootSegment chunkSegment)
      (bsg:successor chunkSegment dataSegment)
      (bsg:semantics chunkSegment "png:chunk:IDAT")
      (bsg:semantics dataSegment "png:chunk-data")
    )
    (and
      (bsg:semantics dataSegment "png:compressed")
      (bsg:type dataSegment "bsg:transcoder")
      (bsg:transcoding dataSegment
        "http://www.dataformats.net/2009/01/25-bsg-ext-ns
          #transcoder:gzip")
    )
  )
)

// F32
(forall
  (var rootSegment)
  (var chunkSegment)
  (var index)

  (exists
    (var compositeSegment)

    (implies
      (and
        (bsg:lastSuccessor rootSegment chunkSegment)
        (bsg:semantics rootSegment "png:root")
        (bsg:semantics chunkSegment "png:chunk")
        (tmp:order chunkSegment index)
        (math:lt 1 index)
      )
      (and
        (bsg:type compositeSegment "bsg:composite")
        (bsg:semantics compositeSegment "png:composite")
      )
    )
  )
)

// F33
(forall
  (var compositeSegment)

  (exists
    (var compressedSegment)

    (implies
      (bsg:semantics compositeSegment "png:composite")
      (and
        (bsg:successor compositeSegment compressedSegment)
        (bsg:semantics compressedSegment "png:compressed")
        (bsg:type compressedSegment "bsg:transcoder")
      )
    )
  )
)

// F34
(forall
  (var compressedSegment)

  (exists
    (var scanlineSegment)

    (implies
      (bsg:semantics compressedSegment "png:compressed")
      (and
        (bsg:successor compressedSegment scanlineSegment)
        (bsg:semantics scanlineSegment "png:scanline")
        (bsg:type scanlineSegment "bsg:transcoder")
        (bsg:transcoding scanlineSegment
          "http://www.dataformats.net/2009/01/25-bsg-ext-ns
            #transcoder:scanline")
      )
    )
  )
)

// F35
(forall
  (var scanlineSegment)

  (exists
    (var pixelSegment)

    (implies
      (bsg:semantics scanlineSegment "png:scanline")
      (and
        (bsg:successor scanlineSegment pixelSegment)
        (bsg:semantics pixelSegment "png:pixels")
        (bsg:type pixelSegment "bsg:primitive")
      )
    )
  )
)

// F36
(forall
  (var chunkSegment)
  (var dataSegment)
  (var compositeSegment)
  (var index)

  (implies
    (and
      (bsg:successor chunkSegment dataSegment)
      (bsg:semantics chunkSegment "png:chunk:IDAT")
      (bsg:semantics dataSegment "png:chunk-data")
      (bsg:semantics compositeSegment "png:composite")
      (tmp:order chunkSegment index)
    )
    (and
      (bsg:type dataSegment "bsg:fragment")
      (bsg:order dataSegment index)
      (bsg:successor dataSegment compositeSegment)
    )
  )
)

/*
(forall
  (var chunkSegment)
  (var dataSegment)
  (var index)

  (implies
    (and
      (bsg:successor chunkSegment dataSegment)
      (bsg:semantics chunkSegment "png:chunk:IDAT")
      (bsg:semantics dataSegment "png:chunk-data")
      (tmp:order chunkSegment index)
    )
    (and
      (bsg:type dataSegment "bsg:fragment")
      (bsg:order dataSegment index)
    )
  )
)
*/

// Rule F37, belonging to the extended fitting set of rules
(forall
  (var chunkSegment)
  (var dataSegment)
  (var dataLength)

  (exists
    (var entrySegment)

    (implies
      (and
        (bsg:successor chunkSegment dataSegment)
        (bsg:semantics chunkSegment "png:chunk:PLTE")
        (bsg:semantics dataSegment "png:chunk-data")
        (bsg:length dataSegment dataLength)
        (math:lte 24 dataLength)
      )
      (and
        (bsg:type dataSegment "bsg:structure")
        (bsg:firstSuccessor dataSegment entrySegment)
        (bsg:type entrySegment "bsg:primitive")
        (bsg:semantics entrySegment "png:palette-entry")
        (bsg:length entrySegment 24)
      )
    )
  )
)

// F38, belonging to the extended fitting set of rules
(forall
  (var chunkSegment)
  (var dataSegment)
  (var dataLength)
  (var previousEntrySegment)
  (var previousEntryEnd)
  (var remainingLength)

  (exists
    (var entrySegment)

    (implies
      (and
        (bsg:successor chunkSegment dataSegment)
        (bsg:successor dataSegment previousEntrySegment)

        (bsg:semantics chunkSegment "png:chunk:PLTE")
        (bsg:semantics dataSegment "png:chunk-data")
        (bsg:semantics previousEntrySegment
          "png:palette-entry")

        (bsg:length dataSegment dataLength)
        (bsg:end previousEntrySegment previousEntryEnd)
        (math:sum previousEntryEnd remainingLength dataLength)
        (math:lte 24 remainingLength)
      )
      (and
        (bsg:leads previousEntrySegment entrySegment)
        (bsg:type entrySegment "bsg:primitive")
        (bsg:semantics entrySegment "png:palette-entry")
        (bsg:length entrySegment 24)
      )
    )
  )
)
```

Appendix B

BSG Reasoning results of PNG ruleset

File name	# reasoning steps	Completeness of BSG instance
BASI0G01.PNG	86	1.0
BASI0G02.PNG	86	1.0
BASI0G04.PNG	86	1,0
BASI0G08.PNG	86	1,0
BASI0G16.PNG	86	1,0
BASI2C08.PNG	86	1,0
BASI2C16.PNG	86	1,0
BASI3P01.PNG	100	0,9545454545454546
BASI3P02.PNG	114	0,9222797927461139
BASI3P04.PNG	114	0,8532110091743119
BASI3P08.PNG	100	0,49705304518664045
BASI4A08.PNG	86	1,0
BASI4A16.PNG	86	1,0
BASI6A08.PNG	86	1,0
BASI6A16.PNG	86	1,0
BASN0G01.PNG	86	1,0
BASN0G02.PNG	86	1,0
BASN0G04.PNG	86	1,0
BASN0G08.PNG	86	1,0
BASN0G16.PNG	86	1,0
BASN2C08.PNG	86	1,0
BASN2C16.PNG	86	1,0

Table B.1: Results of applying PNG data format rules to the PNG Test Suite (1/5).

File name	# reasoning steps	Completeness of BSG instance
BASN3P01.PNG	100	0,9464285714285714
BASN3P02.PNG	114	0,8972602739726028
BASN3P04.PNG	114	0,7777777777777778
BASN3P08.PNG	100	0,40279937791601866
BASN4A08.PNG	86	1,0
BASN4A16.PNG	86	1,0
BASN6A08.PNG	86	1,0
BASN6A16.PNG	86	1,0
BGAI4A08.PNG	86	1,0
BGAI4A16.PNG	86	1,0
BGAN6A08.PNG	86	1,0
BGAN6A16.PNG	86	1,0
BGBN4A08.PNG	100	0,9857142857142858
BGGN4A16.PNG	100	0,9990990990990991
BGWN6A08.PNG	100	0,9702970297029703
BGYN6A16.PNG	100	0,998262380538662
CCWN2C08.PNG	100	0,9788639365918098
CCWN3P08.PNG	114	0,5045045045045045
CDFN2C08.PNG	114	0,9702970297029703
CDHN2C08.PNG	114	0,9651162790697675
CDSN2C08.PNG	114	0,9482758620689655
CDUN2C08.PNG	114	0,9834254143646409
CH1N3P04.PNG	128	0,6976744186046512
CH2N3P08.PNG	114	0,292817679558011
CM0N0G04.PNG	100	0,976027397260274
CM7N0G04.PNG	100	0,976027397260274
CM9N0G04.PNG	100	0,976027397260274
CS3N2C16.PNG	100	0,985981308411215
CS3N3P08.PNG	114	0,6640926640926641
CS5N2C08.PNG	100	0,9838709677419355
CS5N3P08.PNG	114	0,6346863468634686
CS8N2C08.PNG	86	1,0
CS8N3P08.PNG	100	0,625
CT0N0G04.PNG	86	1,0
CT1N0G04.PNG	170	0,4356060606060606
CTZN0G04.PNG	170	0,4581673306772908
F00N0G08.PNG	72	1,0
F00N2C08.PNG	72	1,0
F01N0G08.PNG	72	1,0
F01N2C08.PNG	72	1,0

Table B.2: Results of applying PNG data format rules to the PNG Test Suite (2/5).

File name	# reasoning steps	Completeness of BSG instance
F02N0G08.PNG	72	1,0
F02N2C08.PNG	72	1,0
F03N0G08.PNG	72	1,0
F03N2C08.PNG	72	1,0
F04N0G08.PNG	72	1,0
F04N2C08.PNG	72	1,0
G03N0G16.PNG	86	1,0
G03N2C08.PNG	86	1,0
G03N3P04.PNG	100	0,8598130841121495
G04N0G16.PNG	86	1,0
G04N2C08.PNG	86	1,0
G04N3P04.PNG	100	0,863013698630137
G05N0G16.PNG	86	1,0
G05N2C08.PNG	86	1,0
G05N3P04.PNG	100	0,8543689320388349
G07N0G16.PNG	86	1,0
G07N2C08.PNG	86	1,0
G07N3P04.PNG	100	0,855072463768116
G10N0G16.PNG	86	1,0
G10N2C08.PNG	86	1,0
G10N3P04.PNG	100	0,8598130841121495
G25N0G16.PNG	86	1,0
G25N2C08.PNG	86	1,0
G25N3P04.PNG	100	0,8604651162790697
OI1N0G16.PNG	86	1,0
OI1N2C16.PNG	86	1,0
OI2N0G16.PNG	101	1,0
OI2N2C16.PNG	101	1,0
OI4N0G16.PNG	129	1,0
OI4N2C16.PNG	129	1,0
OI9N0G16.PNG	1389	1,0
OI9N2C16.PNG	3279	1,0
PP0N2C16.PNG	100	0,3264033264033264
PP0N6A08.PNG	100	0,2078239608801956
PS1N0G08.PNG	100	0,1015572105619499
PS1N2C16.PNG	100	0,19134673979280925
PS2N0G08.PNG	100	0,06407518154634771
PS2N2C16.PNG	100	0,1253493013972056
S01I3P01.PNG	114	0,9469026548672567
S01N3P01.PNG	114	0,9469026548672567

Table B.3: Results of applying PNG data format rules to the PNG Test Suite (3/5).

File name	# reasoning steps	Completeness of BSG instance
S02I3P01.PNG	114	0,9473684210526315
S02N3P01.PNG	114	0,9478260869565217
S03I3P01.PNG	114	0,923728813559322
S03N3P01.PNG	114	0,925
S04I3P01.PNG	114	0,9285714285714286
S04N3P01.PNG	114	0,9256198347107438
S05I3P02.PNG	114	0,9104477611940298
S05N3P02.PNG	114	0,9069767441860465
S06I3P02.PNG	114	0,916083916083916
S06N3P02.PNG	114	0,9083969465648855
S07I3P02.PNG	114	0,8993288590604027
S07N3P02.PNG	114	0,8913043478260869
S08I3P02.PNG	114	0,8993288590604027
S08N3P02.PNG	114	0,8920863309352518
S09I3P02.PNG	114	0,8979591836734694
S09N3P02.PNG	114	0,8951048951048951
S32I3P04.PNG	114	0,8816901408450705
S32N3P04.PNG	114	0,8403041825095057
S33I3P04.PNG	114	0,8909090909090909
S33N3P04.PNG	114	0,8723404255319149
S34I3P04.PNG	114	0,8796561604584527
S34N3P04.PNG	114	0,8306451612903226
S35I3P04.PNG	114	0,8947368421052632
S35N3P04.PNG	114	0,8757396449704142
S36I3P04.PNG	114	0,8820224719101124
S36N3P04.PNG	114	0,8372093023255814
S37I3P04.PNG	114	0,8931297709923665
S37N3P04.PNG	114	0,875
S38I3P04.PNG	114	0,8823529411764706
S38N3P04.PNG	114	0,8285714285714286
S39I3P04.PNG	114	0,9
S39N3P04.PNG	114	0,8806818181818182
S40I3P04.PNG	114	0,8823529411764706
S40N3P04.PNG	114	0,8359375
TBBN1G04.PNG	114	0,9904534606205251
TBBN2C16.PNG	114	0,9939819458375125
TBBN3P08.PNG	128	0,5381205673758865
TBGN2C16.PNG	114	0,9939819458375125
TBGN3P08.PNG	128	0,5381205673758865
TBRN2C08.PNG	114	0,9910913140311804

Table B.4: Results of applying PNG data format rules to the PNG Test Suite (4/5).

File name	# reasoning steps	Completeness of BSG instance
TBWN1G16.PNG	114	0,9965095986038395
TBWN3P08.PNG	128	0,5366931918656057
TBYN3P08.PNG	128	0,5366931918656057
TP0N1G08.PNG	86	1,0
TP0N2C08.PNG	86	1,0
TP0N3P08.PNG	100	0,5366071428571428
TP1N3P08.PNG	114	0,5336322869955157
Z00N2C08.PNG	72	1,0
Z03N2C08.PNG	72	1,0
Z06N2C08.PNG	72	1,0
Z09N2C08.PNG	72	1,0

Table B.5: Results of applying PNG data format rules to the PNG Test Suite (5/5).

Appendix C

BSG RDF/N3 Representation

The following RDF Schema definition is given for the BSG RDF/N3 representation in order to encourage the development of BSG-aware third-party applications.

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:skos="http://www.w3.org/2004/02/skos/core#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:undefined="undefined"
  xmlns:bsg="http://dataformats.net/bsg/1.0/">

    <owl:Class
      rdf:about="http://dataformats.net/bsg/1.0/segment"/>

    <owl:Class
      rdf:about="http://dataformats.net/bsg/1.0/structure">
        <rdfs:subClassOf
          rdf:resource="http://dataformats.net/bsg/1.0/
            segment"/>
    </owl:Class>

    <owl:Class
      rdf:about="http://dataformats.net/bsg/1.0/generic">
        <rdfs:subClassOf
          rdf:resource="http://dataformats.net/bsg/1.0/
            segment"/>
    </owl:Class>

    <owl:Class
      rdf:about="http://dataformats.net/bsg/1.0/source">
        <rdfs:subClassOf
          rdf:resource="http://dataformats.net/bsg/1.0/
            segment"/>
    </owl:Class>

    <owl:Class
    rdf:about="http://dataformats.net/bsg/1.0/transcode">
      <rdfs:subClassOf
        rdf:resource="http://dataformats.net/bsg/1.0/
          segment"/>
  </owl:Class>

  <owl:Class
    rdf:about="http://dataformats.net/bsg/1.0/primitive">
      <rdfs:subClassOf
        rdf:resource="http://dataformats.net/bsg/1.0/
          segment"/>
  </owl:Class>

  <owl:Class
    rdf:about="http://dataformats.net/bsg/1.0/fragment">
      <rdfs:subClassOf
        rdf:resource="http://dataformats.net/bsg/1.0/
          segment"/>
  </owl:Class>

  <owl:Class
    rdf:about="http://dataformats.net/bsg/1.0/composite">
      <rdfs:subClassOf
        rdf:resource="http://dataformats.net/bsg/1.0/
          segment"/>
  </owl:Class>

  <rdf:Property
    rdf:about="http://dataformats.net/bsg/1.0/href">
      <rdfs:range
        rdf:resource="http://dataformats.net/bsg/1.0/
          source"/>
      <rdf:type
        rdf:resource="http://www.w3.org/2002/07/owl
          #ObjectProperty"/>
  </rdf:Property>

  <rdf:Property
    rdf:about="http://dataformats.net/bsg/1.0/start">
      <rdfs:range
        rdf:resource="http://dataformats.net/bsg/1.0/
          segment"/>
      <rdf:type
        rdf:resource="http://www.w3.org/2002/07/owl
          #ObjectProperty"/>
  </rdf:Property>

  <rdf:Property
    rdf:about="http://dataformats.net/bsg/1.0/length">
      <rdfs:range
```

```
          rdf:resource="http://dataformats.net/bsg/1.0/
            segment"/>
        <rdf:type
          rdf:resource="http://www.w3.org/2002/07/owl
            #ObjectProperty"/>
    </rdf:Property>

    <rdf:Property
      rdf:about="http://dataformats.net/bsg/1.0/end">
        <rdfs:range
          rdf:resource="http://dataformats.net/bsg/1.0/
            segment"/>
        <rdf:type
          rdf:resource="http://www.w3.org/2002/07/owl
            #ObjectProperty"/>
    </rdf:Property>

    <rdf:Property
      rdf:about="http://dataformats.net/bsg/1.0/semantics">
        <rdfs:range
          rdf:resource="http://dataformats.net/bsg/1.0/
            segment"/>
        <rdf:type
          rdf:resource="http://www.w3.org/2002/07/owl
            #ObjectProperty"/>
    </rdf:Property>

    <rdf:Property
      rdf:about="http://dataformats.net/bsg/1.0/encoding">
        <rdfs:range
          rdf:resource="http://dataformats.net/bsg/1.0/
            primitive"/>
        <rdf:type
          rdf:resource="http://www.w3.org/2002/07/owl
            #ObjectProperty"/>
    </rdf:Property>

    <rdf:Property
      rdf:about="http://dataformats.net/bsg/1.0/codec">
        <rdfs:range
          rdf:resource="http://dataformats.net/bsg/1.0/
            transcode"/>
        <rdf:type
          rdf:resource="http://www.w3.org/2002/07/owl
            #ObjectProperty"/>
    </rdf:Property>

    <rdf:Property
      rdf:about="http://dataformats.net/bsg/1.0/predecessor">
        <rdfs:range
```

```
          rdf:resource="http://dataformats.net/bsg/1.0/
            segment"/>
        <rdf:type
          rdf:resource="http://www.w3.org/2002/07/owl
            #ObjectProperty"/>
    </rdf:Property>

    <rdf:Property
      rdf:about="http://dataformats.net/bsg/1.0/successor">
        <rdfs:range
          rdf:resource="http://dataformats.net/bsg/1.0/
            segment"/>
        <rdf:type
          rdf:resource="http://www.w3.org/2002/07/owl
            #ObjectProperty"/>
    </rdf:Property>
</rdf:RDF>
```

Bibliography

[Abr07a] Stephen Abrams. Global Digital Format Registry (GDFR) Classification v.1.05. Available online at http://www.gdfr.info/docs.html, November 2007.

[Abr07b] Stephen Abrams. Global Digital Format Registry (GDFR) Format Model and Relationships v.1.0.7. Available online at http://www.gdfr.info/docs.html, November 2007.

[ACE+97] O. Avaro, P. A. Chou, Alexandros Eleftheriadis, C. Herpel, C. Reader, and J. Signes. The MPEG-4 Systems and Description Languages: A Way Ahead in Audio Visual Information Representation. *SP:IC*, 9(4):385–431, May 1997.

[AF05] Caroline Arms and Carl Fleischhauer. Digital Formats: Factors for Sustainability, Functionality, and Quality. In *IS&T*, Washington, DC, 2005. The Society for Imaging Science and Technology.

[AG08] Stephen Abrams and Andrea Goethals. Global Digital Format Registry (GDFR) Data Model v.5.0.14. Available online at http://www.gdfr.info/docs.html, May 2008.

[Arm00] Caroline Arms. Keeping Memory Alive: Practices for Preserving Digital Content at the National Digital Library Program of the Library of Congress. *RLG DigiNews*, 4, 2000.

[AS03] Stephen L. Abrams and David Seaman. Towards a Global Digital Format Registry. In *World Library and Information Congress: 69th IFLA GeneralConference and Council*, Berlin, August 2003.

[Bac02] Godmar Back. DataScript - A specification and scripting language for binary data. In *Proceedings of Generative Programming and Component Engineering (GPCE 2002)*, volume 2487 of *Lecture Notes in Computer Science*, pages 66–77, 2002.

[Ben73] Charles H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(2):525–532, 1973.

[BHST08] Sebastian Beyl, Volker Heydegger, Jan Schnasse, and Manfred Thaller. Final XCDL Specification. Project "Planets", Sub-Project PC/2, Deliverable D7, May 2008.

[BPVdWK06] Ian Shaw Burnett, Fernando Peirera, Rik Van de Walle, and Rob Koenen, editors. *The MPEG-21 Book.* John Wiley and Sons Ltd, 2006.

[Bro05] Adrian Brown. PRONOM 4 Information Model. Available online at http://www.nationalarchives.gov.uk-/aboutapps/fileformat/pdf/pronom_4_info_model.pdf, January 2005.

[CCS00] CCSDS. *The Data Description Language EAST Specification.* Consultative Committee for Space Data Systems (CCSDS), 2000.

[CCS02] *Reference Model for an Open Archival Information System (OAIS)*, volume Blue Book. Consultative Committee for Space Data Systems, January 2002.

[CFP+04] Corinna Cortes, Kathleen Fisher, Daryl Pregibon, Anne Rogers, and Frederick Smith. Hancock: A language for analyzing transactional data streams. *ACM Transactions on Programming Languages and System (TOPLAS)*, 26(2):301–338, March 2004.

[CGT89] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, March 1989.

[Cho59] Noam Chomsky. On certain formal properties of grammars. *Journal of Information and Control*, 2(2):137–167, June 1959.

[CSF+08] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, May 2008.

[Dev03] Sylvain Devillers. An Extension of BSDL for Multimedia Bitstream Syntax Description. In *Euro-Par*, pages 1216–1223, 2003.

[DN] Wesley De Neve. Description-driven media resource adaptation.

[DNVDDS+06] Wesley De Neve, Davy Van Deursen, Davy De Schrijver, Sam Lerouge, Koen De Wolf, and Rik Van de Walle. BFlavor: A harmonized approach to media resource adaptation inspired by MPEG-21 BSDL and XFlavor. *EURASIP Signal Processing: Image Communication*, 21(10):862 –889, 11 2006.

[Duc03] Portable Network Graphics (PNG) Specification (Second Edition): Information technology Ů– Computer graphics and image processing Ů– Portable Network Graphics (PNG): Functional specification. ISO/IEC 15948:2003 (E), November 2003.

[Dym91] Marc Dymetman. Inherently Reversible Grammars, Logic Programming and Computability. In *Proc. ACL Workshop on Reversible Grammar in Natural Language Processing*, pages 20–30, 1991.

[EH02] Alexandros Eleftheriadis and Danny Hong. XFlavor: bridging bits and objects in media representation. In *Proceedings of the IEEE International Conference on Multimedia and Expo, 2002*, volume 1, pages 773–776, 2002.

[Ele95] Alexandros Eleftheriadis. A Syntactic Description Language for MPEG-4. Contribution ISO/IEC JTC1/SC29/WG11 MPEG95/M0546, November 1995.

[Ele96] Alexandros Eleftheriadis. The Benefits of Using MSDL-S for Syntax Description. Contribution ISO/IEC JTC1/SC29/WG11 MPEG96/M1555, November 1996.

[Ele97] Alexandros Eleftheriadis. Flavor: A Language for Media Representation. In *Proceedings of the 5th ACM International Conference on Multimedia (MM97)*, pages 1–9, New York, NY, USA, 1997. ACM Press.

[ETS05] ETSI TS 101 349 V8.27.0 (2005-09): Digital cellular telecommunications system (Phase 2+); General Packet Radio Service (GPRS); Mobile Station (MS) - Base Station System (BSS) interface; Radio Link Control/ Medium Access Control (RLC/-MAC) protocol. ETSI 3GPP, September 2005.

[ETS10] ETSI TS 124 007 V9.0.0 (2009-03): Digital cellular telecommunications system (Phase 2+); Universal Mobile Telecommunications System (UMTS); LTE; Mobile radio interface signalling layer 3; General Aspects. ETSI 3GPP, January 2010.

[FG05] Kathleen Fisher and Robert Gruber. PADS: A Domain-Specific Language for Processing Ad Hoc Data. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 295–304, 2005.

[FMW06] Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. The Next 700 Data Description Languages. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL 2006)*, 2006.

[Fra97] Michael P. Frank. The R Programming Language and Compiler. Mit reversible computing project memo m8, MIT, July 1997.

[HBSM08] Michael Hartle, Arsene Botchak, Daniel Schumann, and Max Mühlhäuser. A Logic-based Approach to the Formal Description of Data Formats. In *Proceedings of The Fifth International Conference on Preservation of Digital Objects (iPRES)*, pages 292–299, London, United Kingdom, September 2008. The British Library.

[HE08] Danny Hong and Alexandros Eleftheriadis. XFlavor: providing XML features in media representation. *Multimedia Tools and Applications*, 39(1):101–116, August 2008.

[Hed04] Shawn Hedman. *A First Course in Logic: An Introduction to Model Theory, Proof Theory, Computability, and Complexity.* Oxford University Press, 2004.

[HFS+09] Michael Hartle, Andreas Fuchs, Marcus Ständer, Daniel Schumann, and Max Mühlhäuser. Data Format Description and its Applications in IT Security. *International Journal On Advances in Security*, 2(1):90–111, 2009. http://www.iariajournals.org/security/.

[HMT+08] Michael Hartle, Friedrich-Daniel Möller, Slaven Travar, Benno Kröger, and Max Mühlhäuser. Using Bitstream Segment Graphs for Complete Data Format Instance Description. In José Cordeiro, Boris Shishkov, Alphes Kumar Ranchordas, and Markus Helfert, editors, *Proceedings of The Third International Conference on Software and Data Technologies (ICSOFT)*, pages 198–205, Porto, Portugal, August 2008. Institute for Systems and Technologies of Information, Control and Communication.

[HSB+08] Michael Hartle, Daniel Schumann, Arsene Botchak, Erik Tews, and Max Mühlhäuser. Describing Data Format Exploits using Bitstream Segment Graphs. In *Proceedings of The Third International Multi-Conference on Computing in the Global Information Technology (ICCGI)*, pages 119–124, Athens, Greece, March 2008. IARIA, IEEE Press, New York, NY.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[ISO00] ISO/IEC 13818-1:2000: Information technology – Generic coding of moving pictures and associated audio information: Systems, 2000.

[ISO05a] ISO/IEC 14496-12:2005: Information technology – Coding of audio-visual objects – Part 12: ISO base media file formta, 2005.

[ISO05b] ISO/IEC 7816-4:2005: Identification cards – Integrated circuit cards – Part 4: Organization, security and commands for interchange. ISO/IEC, 2005.

[IT95] ITU-T. Recommendation H.262 (07/95) — Information Technology - Generic Coding of Moving Pictures and Associated Audio Information: Video. ITU-T, Geneva, July 1995.

[IT97] ITU-T. Recommendation X.680 (12/97) — Abstract Syntax Notation One (ASN.1): Specification of Basic Notation. ITU-T, Geneva, December 1997.

[IT01] ITU-T. Recommendation X.693 (12/01) — ASN.1 Encoding Rules: XML Encoding Rules (XER), December 2001.

[IT02a] ITU-T. 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, July 2002.

[IT02b] ITU-T. Recommendation X.691 (07/02) — ASN.1 Encoding Rules: Specification of Packed Encoding Rules (PER). ITU-T, Geneva, July 2002.

[IT02c] ITU-T. Recommendation X.692 (03/02) — ASN.1 Encoding Rules: Specification of Encoding Control Notation (ECN). ITU-T, Geneva, March 2002.

[IT06] ITU-T. Recommendation H.323 (06/06) — Packed-based multimedia communications systems. ITU-T, Geneva, June 2006.

[Lan61] Rolf Landauer. Irreversibility and Heat Generation in the Computing Process. *IBM Journal of Research and Development*, 5(3):183–191, 1961.

[Lar99] John Larmouth. *ASN.1 Complete*. Morgan Kaufmann, 1999.

[LC04] Lei Li and Krishnendu Chakrabarty. On Using Exponential-Golomb Codes and Subexponential Codes for System-on-a-Chip Test Data Compression. *Journal of Electronic Testing: Theory and Applications*, 20:667–670, 2004.

[LDT01] J. Larmouth, O. Dubuisson, and J. Thorpe. Application of the ASN.1 specification technique to the Bluetooth Service Discovery Protocol. In *In Proceedings of the ACM Symposium on Mobile Ad Hoc Networking & Computing (MobiHoc)*, Long Beach, California, October 2001.

[Lec63] Yves Lecerf. Machines de Turing réversibles, Récursive insolubilité en $n \in N$ de lťéquation $u = \theta^n$, où θ est un “isomorphisme de codes”. *Comptes Rendus*, 257:2597–2600, 1963.

[Lor01] Raymond A. Lorie. Long Term Preservation of Digital Information. In *In Proceedings of the Joint Conference on Digital Libraries (JCDL 2001)*, pages 346–352, 2001.

[Mar07] Marsu. Photoshop CS2/CS3, Paint Shop Pro 11.20 .PNG File Buffer Overflow. http://milw0rm.com/exploits/3812, 2007.

[MC00] Peter J. McCann and Satish Chandra. PacketTypes: Abstract specification of network protocol messages. In *Proceedings of ACM Conference of Special Interest Group on Data Communications (SIGCOMM)*, pages 321–333, August 2000.

[MC03] James D. Myers and Alan Chappell. Binary Format Description (BFD) Language, 2003.

[Ock98] John Marc Ockerbloom. *Mediating Among Diverse Data Formats*. PhD thesis, Carnegie Mellon Computer Science, 1998.

[Ock06] John Mark Ockerbloom. The Next Mother Lode for Large-scale Digitization? Historic Serials, Copyrights, and Shared Knowledge. Available online at http://repository.upenn.edu/cgi/viewcontent.cgi?article=1071&context= April 2006.

[PHB+10] Alan Powell, Steve Hanson, Mike Beckerle, Martin Westhead, Geoff Judd, and Robert E. McGrath. Data Format Description Language (DFDL) v1.0 Core Specification (Internal Committee Working Document). Available online at `http://forge.gridforum.org/sf/docman/do/downloadDocument/pro` January 2010.

[PHH+03] Gabriel Panis, Andreas Hutter, Jörg Heuer, Hermann Hellwagner, Harald Kosch, Christian Timmerer, Sylvain Devillers, and Myriam Amielh. Bitstream Syntax Description: A Tool for Multimedia Resource Adaptation within MPEG-21. *EURASIP Signal PRocessing: Image COmmunication Journal*, 18(8):721–747, September 2003.

[RG99] Seamus Ross and Ann Gow. *Digital Archaeology: Rescuing Neglected and Damaged Data Resources*. Library Information Technology Centre, South Bank University, February 1999.

[RH05] Seamus Ross and Margaret Hedstrom. Preservation research and sustainable digital libraries. *International Journal on Digital Libraries*, 5(4):317–324, 2005.

[Rot99] Jeff Rothenberg. Ensuring the Longevity of Digital Information. Available online at http://www.clir.org/pubs/archives/ensuring.pdf, February 1999.

[SHC08] Jan Schnasse, Volker Heydegger, and Elona Chudobkaite. Final XCEL Specification. Project "Planets", Sub-Project PC/2, Deliverable D8, July 2008.

[Sip97] Michael Sipser. *Introduction to the Theory of Computation.* PWS Publishing, 1997.

[UDF09] UDFR. Unified Digital Formats Registry (UDFR) Proposal and roadmap. Available online at http://gdfr.info/udfr_docs/Unified_Digital_Formats_Registry.pdf, March 2009.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II.* Computer Science Press, 1989.

[VCE03] Anthony Vetro, Charilaos Christopoulos, and Touradj Ebrahimi. From the guest editors: Universal Media Access. *IEEE Signal Processing Magazine*, 20(2):16–16, March 2003.

[VDDNDSVdW08] Davy Van Deursen, Wesley De Neve, Davy De Schrijver, and Rik Van der Walle. gBFlavor: a new tool for fast and automatic generation of generic bitstream syntax descriptions. *Multimedia Tools Appl.*, 40(3):453–494, 2008.

[Vie95] Carlin J. Vieri. Pendulum: A reversible computer architecture. Master's thesis, MIT Artificial Intelligence Laboratory, 1995.

[vS98] Willem van Schaik. PngSuite - the official set of PNG test images, December 1998. http://www.schaik.com/pngsuite/pngsuite.html, last accessed 2008-01-02.

[WAKC97] Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Serra S. Christopher. The Zephyr Abstract Syntax Description Language. In *Proceedings of the Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997.

[Wes02] Martin Westhead. BinX - The Binary XML Description Language. Technical report, epcc, April 2002.

[Wet98] Michael Wettengel. *German Unification and Electronic Records*, chapter 18, pages 265–276. Oxford University Press, 1998. ISBN 0198236336.

[WHK97] M. Wahl, T. Howes, and S. Kille. Lightweight Directory Access Protocol (v3). RFC 2251, December 1997.

[YG07] Tetsuo Yokoyama and Robert Glück. A Reversible Programming Language and its Invertible Self-Interpreter. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 144–153. ACM, New York, USA, 2007.