Gathering Evidence

Model-Driven Software Engineering in Automated Digital Forensics

Gathering Evidence
Model-Driven Software Engineering in Automated Digital Forensics

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. D.C. van den Boom
ten overstaan van een door het college voor promoties
ingestelde commissie,
in het openbaar te verdedigen in de Agnietenkapel
op donderdag 9 januari 2014, te 10.00 uur

door

Jeroen van den Bos

geboren te Delft

Promotiecommissie

Promotor:	Prof. dr. P. Klint
Co-promotor:	Dr. T. van der Storm
Overige leden:	Prof. dr. J.A. Bergstra Dr. ing. Z.J.M.H. Geradts Prof. dr. ir. C.T.A.M. de Laat Prof. dr. R. Lämmel Prof. dr. R.F. Paige Prof. dr. M. de Rijke

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

Centrum Wiskunde & Informatica

The work in this thesis has been carried out at Centrum Wiskunde & Informatica (CWI) in cooperation with the Netherlands Forensic Institute (NFI), under the auspices of the research school IPA (Institute for Programming research and Algorithmics)

Contents

Preface

In the summer of 2002 I was invited to interview for the position of software engineer at the Digital Technology department at the Netherlands Forensic Institute (NFI). I ended up as one of the first two software engineers to be hired, and we set up a software engineering-group within the department, dedicated to developing forensic software. Apart from dealing with the engineering challenges that are the subject of this thesis, I was encouraged to develop and spread knowledge in many ways. This included supervising students, teaching users to apply our tools, writing publications and attending conferences.

It didn't take very long for me to realize that I would like to be involved in the necessary innovations around what we now refer to as *automated digital forensics*. However, as a self-taught programmer without any formal training, I figured it would be difficult to participate at the forefront of digital forensics technology. So I decided to pursue the necessary education, *on the side*. At least, that's how I thought I was going to do it.

I enrolled in a part-time program to obtain a bachelor's degree in computer science, managing to complete nearly all courses in the first year and spending my free time in the second year writing a thesis. Some courses exposed me to work in the area of programming languages by members of the SWAT-group at Centrum Wiskunde & Informatica (CWI). I discovered this research group had its own master's program in software engineering at the University of Amsterdam, so I decided to enroll into the full-time program, taking off almost a full year from my work at the NFI.

During my master's I got interested in program transformation and was allowed to do my thesis research in the SWAT-group at CWI. I saw some clear opportunities to apply the techniques I had been working with to improve digital forensics. Around this time I started wondering whether another step would be possible, such as some kind of co-operation between NFI and CWI to allow me to really pursue some of those ideas. I decided to propose it. This PhD thesis is the result of what happened to that proposal.

Acknowledgements

I owe a debt of gratitude to my promotor Paul Klint, for organizing many important steps that have allowed me to get to the completion of this thesis. Paul always displays a combination of determination and patience that he shows to be crucial for success. Although these traits are exceptionally difficult to reproduce, I will try.

When I started my research at CWI, I expected my co-promotor Tijs van der Storm to help me get up to speed in research and writing. Instead, Tijs has been an incredible mentor, teaching me how to think critically and express myself precisely. I am greatly indebted to him for this, and hope for more collaboration in the future.

I am also grateful to the members of my PhD committee: Jan Bergstra, Zeno Geradts, Cees de Laat, Ralf Lämmel, Richard Paige, and Maarten de Rijke, for their willingness to read my thesis and provide me with thoughtful feedback.

In the past four years when I was doing my research at SWAT, the group transitioned in leadership from Paul Klint to Jurgen Vinju. It is telling that even though the group was built and then lead by Paul for a very long time, nobody seemed to worry about this transition. Working with Jurgen is a pleasure, as he manages to combine strategy and vision with a personal interest in everybody he works with.

I am lucky to have had very supportive co-workers over the years at both the CWI and NFI, of which many have turned into friends along the way. Bas Basten and I have spent many afternoons discussing all conceivable subjects, usually starting out at a technical detail of whatever we were working on and then slowly branching out towards whatever came up.

Mark Hills has been a source of inspiration over the years as I regularly struggled with the complexities of implementing compilers. His deep knowledge and meticulousness has saved me many times. My only regret is that we never got him to try horse meat, even though it really does go wonderfully with pindasaus.

All my papers, and this entire thesis especially, would look a lot less attractive if it weren't for the help of Davy Landman, who manages to turn any task into an epic engineering endeavor. We seem to have a fully compatible sense of humor, which has greatly enriched my time at SWAT.

For almost the entire four years, I have shared a room with Bert Lisser, who has an almost encyclopedic knowledge of the history of CWI. I will miss our early morning coffee breaks.

I will also miss the other (former) members of SWAT and the software engineering program that I worked with: Ali Afroozeh, Magiel Bruntink, Hans Dekkers, Jan van Eijck, Mike Godfrey, Paul Griffioen, Pablo Inostroza Valdera, Anastasia Izmaylova, Arnold Lankamp, Robert van Liere, Atze van der Ploeg, Riemer van Rozen, Alexander Serebrenik, Ashim Shahi, Floor Sietsma, Sunil Simon, Michael Steindorfer, Yanjing Wang, and Vadim Zaytsev.

Which brings me to the NFI, where I have been met with similar supportiveness and many friendships. I owe a great deal to Lotte Smelik, both for encouraging me to pursue my research interests by trying to set up a co-operation with CWI as well as for approving the final proposal.

Shortly after I started working on my thesis, Erica Rietveld became department manager and I was somewhat worried whether she would fully support my project. Instead, not only was she supportive, she turned out to understand many of the intricacies of what my research was about, allowing me to focus on it fully.

At the software engineering group, the first seeds of this research were planted in discussions with Arjen van de Wetering, who succeeded me as team lead in 2005. Over the years Arjen has been a great source of inspiration and discussion, even after he left the NFI to work in model-driven engineering.

Leon Aronson combines vision with realism, which often leads to great ideas and occasionally to hilarious insights. He has been a great support over the years, both professionally and personally. Ewald Snel always manages to challenge me whenever I take a couple of technical truths for granted. Stefan Nelwan has been very supportive, helping me to set out a course for the future.

All members of the software engineering-group (past and present) have contributed in various ways to this thesis, for which I am grateful: Erik Aleman, Pelle Barens, Jörgen Bodde, Roel van Dijk, Wendy van Dijk, Michel Frenaij, Mirelle Goos, Jeroen de Jong, Arent de Jongh, Bas Knopper, Sander Kruseman, John Langezaal, Arjen Meijer, Robert Moro, Jolijn Posthuma, Martijn Ras, Jan-Willem Renema, and Allard Siemelink.

I am grateful for the feedback I have received over the years at the NFI, most notably from: Raoul Bhoedjang, Erwin van Eijk, Zeno Geradts, and Ronald van der Knijff.

These past years have been exceptionally busy, which have lead me to neglect my best friends, Frans Bouma and Farid Jabli. Nonetheless, they have managed to greatly influence my thinking and research.

Mam en pap, jullie steun en liefde voor mij en mijn gezin zijn een enorme verrijking van ons leven. Dit proefschrift is ook van jullie.

Finally, from the very first ideas to the often difficult deadlines, my wife Nicky has always supported me with advice, understanding, and love. I owe everything to her.

Jeroen van den Bos

Pijnacker, 21 november 2013

Gathering Evidence

Model-Driven Software Engineering in Automated Digital Forensics

It is a capital mistake to theorise before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts.

– Arthur Conan Doyle, *The Adventures of Sherlock Holmes* (1892)

Part I

Overview and Analysis

Highlights:

- Domain analysis of the characteristics of file formats.
- Domain analysis of the software requirements of data recovery tools.

CHAPTER 1

Introduction

In 2008, the mother of Caylee Anthony was accused of murdering her daughter. A key piece of evidence brought forward by the prosecution was that a website discussing the use of chloroform was visited 84 times on a computer the defendant had access to.

This fact was discovered by an automated analysis of the data stored on the devices related to the investigation. The analysis was performed by a digital forensics software tool that recovers and analyzes potentially relevant information.

However, the tool's designer discovered that his software contained an error which lead to reporting incorrect information. A subsequent redesign of the software, which allowed the relevant internet history file to be correctly decoded, revealed that the website had only been visited once. The error was attributed to complexities in the file format that the computer's web browser used to store its history [Alv11].

As shown by this case, the impact a single piece of software can have in a digital forensic investigation is huge. At the same time, the large amount of information, spread across many locations and all encoded in different and evolving file formats, presents a significant challenge to investigators.

The only scalable solution is to automate the majority of this work: making secure copies of data, recovering information in many shapes and aggregating and visualizing the information for analysis on a higher level than individual items. All this work has to be executed by software that is forensically sound, which refers to processing data without modifying it, without built-in assumptions with regard to interpretation and exhaustively logging all performed steps.

The specialist nature of this functionality results in digital forensics relying almost entirely on custom software engineering. In addition to the domain-specific functionality, automated digital forensics tools share the same non-functional requirements: high runtime performance, scalability and modifiability:

Runtime performance: Analyses need to be fast. This is a hard requirement due to legal restrictions on, for example, pre-charge detainment of suspects.

Scalability: An exponential increase in processing, storage and networking capacity, along with growing popularity, requires tools to scale up.

Modifiability: Many different, evolving, and emerging file formats, requires constant adaptation of the tools.

Realizing these qualities presents a significant engineering challenge, as they are naturally at odds: all three require the software to be optimized in a different dimension [BCK12]. For runtime performance, optimization depends on algorithm selection and implementation, as well as allocation of functionality to different modules, communication between modules and shared resource use.

For scalability, optimization also depends on allocation of functionality to different modules, but specifically to allow modules to be replaced by others with different capacities. For modifiability, optimization depends on how functionality is divided and how it is implemented, again usually at the level of modules.

This thesis presents an approach to address this challenge in one area of automated digital forensics engineering through the use of model-driven software engineering (MDSE). We have developed the domain-specific language (DSL) Derric, that captures a significant part of the problem space of data recovery applications and is designed specifically to be easy to understand and modify. We achieve this through an analysis of the domain and common changes to such applications.

Derric is used to describe the structure of file formats, such as image and document files. File format descriptions in Derric are declarative and independent of any implementation. The syntax resembles how investigators encounter and use file format information[1], making the descriptions easy to understand and modify.

File format descriptions in Derric are transformed to one or several implementations by a compiler and interpreter. These components encode the design and implementation decisions to achieve high runtime performance and scalability.

We evaluate our approach in several ways. First, we use our solution to build a typical digital forensics tool, called a *file carver* [PM09] and compare it to a set of existing file carvers on a set of existing benchmarks. Second, we implement a set of model transformations to allow the generation of components with different

[1]Common sources are hex editors, source code, standards documents and informal specifications.

runtime performance characteristics, to allow the user to make custom trade-offs to improve scalability. We evaluate these transformations on a custom benchmark.

Third, we perform a set of modifications to programs written in DERRIC in order to evaluate its flexibility in realistic maintenance scenarios. Finally, we construct an integrated development environment to assist in performing maintenance. Additionally, this demonstrates how an MDSE approach allows the construction of domain-specific tool support, that would be difficult to develop otherwise.

1.1 Automated Digital Forensics

At the inaugural Digital Forensics Research Workshop in 2001, the following definition for *digital forensics* was proposed:

> *The use of scientifically derived and proven methods toward the preservation, collection, validation, identification, analysis, interpretation, documentation and presentation of digital evidence derived from digital sources for the purpose of facilitating or furthering the reconstruction of events found to be criminal, or helping to anticipate unauthorized actions shown to be disruptive to planned operations [Pal01].*

ISO and IEEE use the following definition for *software engineering*:

> *The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software [ISO10].*

When it is required, or at least desired, to develop and use software in order to perform digital forensic investigations, these areas intersect. While all digital forensic investigations depend on software, in practice a stage beyond simply using software tools to complete individual tasks is required: *automated digital forensics*.

The *automated* refers to the automatic execution of multiple steps in a digital forensic investigation. This can either mean multiple steps on a single piece of data, such as collection, identification and presentation, or to the processing of multiple pieces of data in a batch, such as recovering millions of files from a hard drive.

Automatic batch processing of data has become a critical requirement in the past decades, as processing capacity, digital storage size as well as network bandwidth have continued to grow exponentially. As a result, in nearly any case, initial investigation of single pieces of data has become intractable [Gar10].

Although there are some differences in terminology (e.g., acquisition [Cas09] versus preservation [Car05]), the partitioning of digital forensic investigations is widely agreed upon. There are three main phases that occur in any digital forensic investigation, regardless of the level of automation:

Acquisition: Consists of all tasks associated with making a secure copy of the data under investigation, so that it can be further investigated independently of the original device and without risk of data loss.

Recovery: Extracting information from the acquired data for further analysis, often at multiple levels, such as a file's metadata, its main content (such as its multimedia or text content), and potentially embedded files.

Analysis: Applying different techniques such as aggregation and visualization in order to answer legal questions. Mostly concerned with reducing the amount of information to digest for investigators.

Each device goes through acquisition only once: when its contents have been securely copied and backed up, only this copied data is used in the next phases. Only when during a successive phase it is discovered or suspected that the acquisition was incorrect or incomplete, acquisition is restarted (but this essentially boils down to a restart of the entire process).

Recovery and analysis may be performed iteratively. For example, under high time pressure, recovery may be reduced in precision to reach the analysis phase quickly. However, once some relevant facts are discovered, a more precise version of the recovery phase may be attempted, in order to find more information. These phases and their relationships are shown in Figure 1.1.

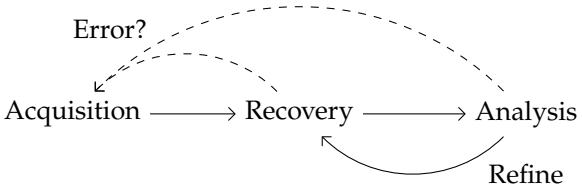

Figure 1.1: The phases of a digital forensic investigation and their relationships.

This partitioning is similar to those in other domains, such as the Extract, Analyze, Synthesize (EASY) paradigm in metaprogramming [KSV09b] and Extract, Transform, Load (ETL) processes in data warehousing [Vas11].

Acquisition

The first step in any digital forensic investigation is to *acquire* a copy of all the relevant data that may need to be investigated. Once a copy is made, it is authenticated by calculating one or more types of cryptographic hashes and backed up to prevent data loss due to errors in subsequent phases.

An important decision to make during this phase is whether to attempt a dead or live acquisition [Car05]. This refers to whether a system is off (dead) or on (live) when its contents is copied.

While turning a device off will reduce the amount of possible outside influences (such as an installed rootkit), it may also revoke access to, for example, a decrypted area on the system. Furthermore, a Faraday cage can be used to prevent potential outside influences during a live acquisition from a device that has wireless networking capabilities [Wil05].

To prevent inadvertent modification of the evidence during copying, a *write blocker* is used. While the actual copying application is most likely designed specifically not to modify its source data, other components such as the computer's operating system or BIOS, may attempt to write to the device. Write blockers exist in two forms: hardware [Nat04] and software [Nat03].

Additionally, not all evidence is encountered on fully functioning computers or devices that are either turned on or off. Hardware may be significantly damaged, in which case parts of it may be repaired or replaced in order to access its contents. Some hardware may be intact, but not accessible due to a legacy or proprietary interface, in which case it may be taken apart (e.g., to access memory chips directly).

Recovery

Once a secure copy of the data under investigation is available, the next step is to *recover* as much information from it as possible. The recovery phase transforms large monolithic blocks of data (e.g., one data stream per device) into information that can be interpreted, such as e-mails, logs, and image and document files.

Most information, is recovered using different kinds of metadata commonly available in acquired data streams. For example, nearly all devices store their data in a file system, that records physical locations for each file in the stream, along with the file's name. Additionally, the metadata itself may be considered relevant information as well, such as time stamps of when or by who a file was created, last accessed or modified.

However, the required metadata is not always readily available. For example, a deleted file's contents and metadata are generally not immediately overwritten (or cleared) when a file is deleted, but instead marked as "available for writing". With knowledge of the data structures of the file system, deleted files that are not yet overwritten can be recovered by looking beyond just the directly referenced links.

Even when the metadata is unavailable, the file's contents may still be present in the data stream. In order to recover it, a content-based approach can be used to attempt to recognize, called *validate*, a file by its internal structure. All content-based recovery approaches are referred to as *file carving* [PM09].

Such a content-based approach can be combined with reconstruction algorithms to recover files that are fragmented (i.e., divided into multiple parts). To prevent a combinatorial explosion, the employed algorithms usually reduce the search space to look for files fragmented in a common and simple pattern (e.g., bifragment gap-carving [Gar07], see also Chapter 2).

Finally, validation also serves an important function even when files were directly recoverable: the metadata is not always correct. The simplest form is an incorrectly named file, such as a JPEG image that has a DOC extension. Other possibilities include concatenated files, where only the first file would be identified (and the concatenated file(s) could be ignored during further analysis).

Files may also be investigated for embedded files. Even though embedding files is a normal practice (such as a JPEG image embedded in a PDF document), it is beneficial for analysis to have all embedded files available separately.

An example overview of how recovered files may relate to their physical storage location is shown in Figure 1.2. It depicts a scenario where a file is embedded inside another file, that is in turn concatenated to another file and then stored in a logical file (i.e., a file from the perspective of the file system and/or operating system). This logical file is then represented as a stream in memory, which is stored in two fragments in the file system, which in turn ends up in three fragments in the solid state memory the file system resides on.

Analysis

Once recovery completes, the process begins to *analyze* the available information. The process can itself consist of two distinct steps: *loading* the information into a database or other information retrieval system and *querying* it for relevant facts. However, since the loading step will potentially make decisions about which facts to include and/or exclude, we consider them both to be part of analysis.

Due to the large amount of information generally involved in a digital forensic investigation, along with the potentially complex legal framework surrounding any conclusion, an analysis consists mostly of reducing the amount of information to digest for a forensic investigator. Additionally, any conclusion will have to be validated manually.

An example of automatic reduction of the amount of information to consider is the use of a hash database of known files [RR06]. For each file, a hash is calculated and then compared to those in the database. If a match is found, such a file can then either be automatically included or excluded (depending on whether the database contains hashes of known relevant or irrelevant files respectively).

The analysis phase is essentially free form, since it consists of any conceivable analysis on large sets of heterogenous information. Because of this, we will refrain from defining fixed characteristics and instead describe some examples.

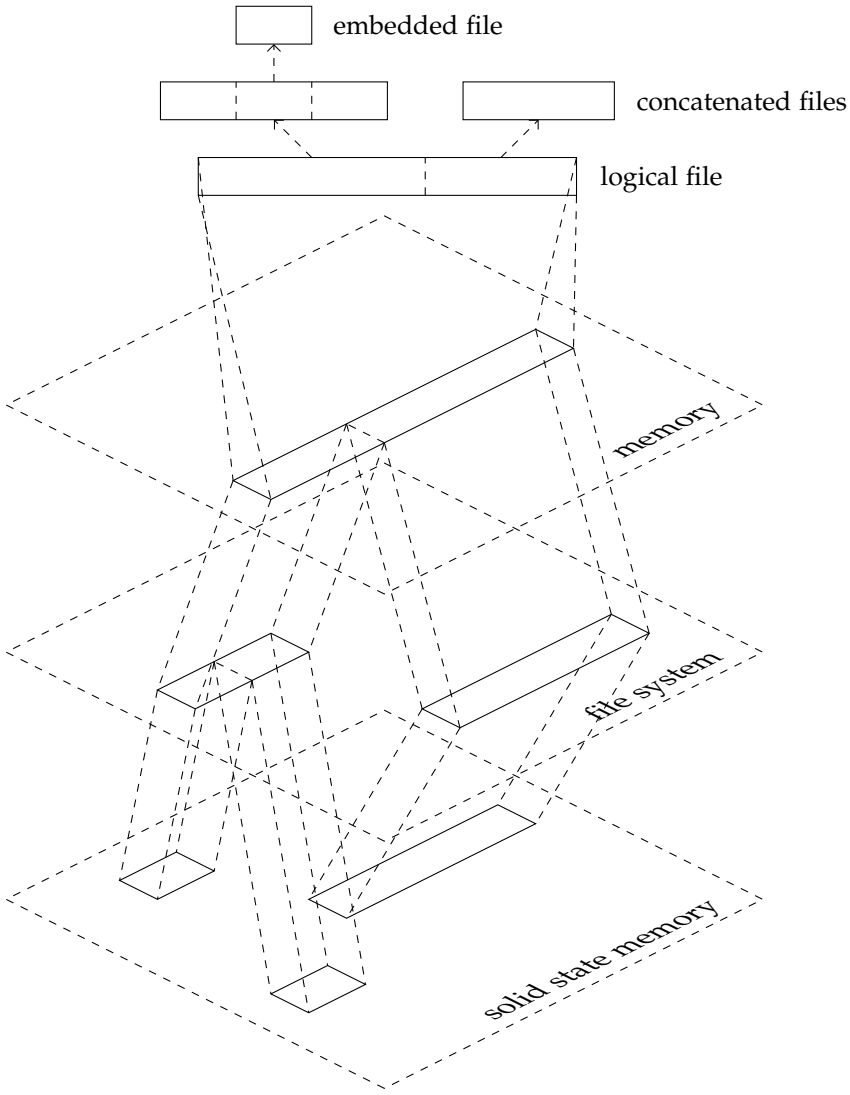

Figure 1.2: The relationship between recovered files and how they are stored.

An example of a relatively simple analysis is to filter all recovered data to collect and present all images of a specific type, so that they can be viewed by an investigator to determine which of the images are relevant to an investigation. While simple, having to manually traverse a file system to look for images, including unpacking

compressed files, will take up a considerable part of an investigator's time.

An example of a relatively complex analysis is a network analysis across recovered information from multiple sources to determine links between suspects, such as whether they have communicated or share large parts of a collection of files. For example, receiving an e-mail from someone does not imply knowing that person, but corresponding for an extended period of time suggests acquaintance.

Finally, an analysis that may lead to a return to the recovery process (as shown in Figure 1.1) is where metadata of image files is examined to determine the likelihood of additional files from a series being present in the acquired data. For example, if images contain a numbering scheme, a small set of missing numbers may indicate that a more time consuming recovery step can yield important results by locating the missing files.

1.2 Model-Driven Software Engineering

Nearly all software is developed using third-generation languages such as C/C++, Java and PHP. They have lead to the development of large libraries of high-level abstractions covering all kinds of domains including many in the software engineering domain itself (such as middleware).

Unfortunately, with the advance of software, hardware and networking capabilities, the complexity of developing applications has nonetheless increased. The main reason for this is that third-generation languages still require high level goals to be expressed in often thousands of lines of code [Sch06].

A potential solution to this problem is the development of *Domain-Specific Languages* (DSLs), or more generally, the application of *Model-Driven Software Engineering* (MDSE) [Béz06]. Two parts of any MDSE approach are the following:

Direct Representation: A custom notation to allow the expression of the solution at a high level of abstraction. These descriptions are not just documentation, but first-class entities in the development process.

Automation: The implementation is automatically generated from high-level models expressed in the DSL. This means that the semantic gap between high-level expression and low-level implementation is crossed automatically.

These parts were described in early work on MDA (an early manifestation of MDSE) [BBI+04] along with standards to guarantee interoperability between different technical solutions.

A key issue is how to determine whether the application of MDSE is beneficial given an engineering challenge. Investment in designing a DSL and the related implementation is significant, given that it requires extensive knowledge of both

the application domain (in order to design the notation) and modeling/language technology (in order to implement the automation) [MHS05].

The goals underlying any decision to use MDSE are usually to make an organization's software engineering activities more economical as well as allow direct participation in the software engineering process by end users [MHS05].

A large number of benefits are ascribed to using MDSE, including increasing maintainability [DK98], reliability [Spi01] and reusability [Kru92]. Success factors observed in practice include increased maintainability [BJMH02, KSV10] and reliability, as well as shorter time-to-market and reduced development costs [HPD09].

Many open questions still remain on the use of MDSE, especially in relation to how to apply existing tools in practice [PV12].

Direct Representation

Developing a DSL to solve or specify problems in a specific domain has been a practice throughout the entire history of computer science [DKV00]. For example, classic general-purpose programming languages such as Cobol and Fortran were originally designed to solve problems in a specific domain.

A graphical DSL, or visual language, can be used to express models and programs using a rich set of notational constructs, such as shapes, connectors, distance and orientation. As a result, graphical languages have a syntactically dense layout: they allow the use of positioning to express an almost infinite amount of relationships [Ray91].

Although graphical notations are often thought to be easier to understand and use for beginners, the large amount of different layouts they enable may actually make them more difficult for these users [Pet95]. Textual languages can be considered highly constrained graphical languages, which may be an advantage to both the user and implementer.

Apart from the constraints on a textual language, they also allow the use of existing programming tools such as version control systems and program comparison tools without requiring specific extensions [XS05]. In practice however, there are many benefits to developing custom solutions for such tasks such (e.g., model comparison [KPP06]).

Implementation Patterns

Application frameworks and libraries can be considered domain-specific languages, since their interfaces provide a kind of "language" to express concepts in a specific domain. This idea can be extended in (usually) functional programming languages by using specific patterns to facilitate the use of a domain-specific syntax. Essen-

tially, such an *internal DSL* is an application framework, but with a different flavour to it [Fow10].

This implementation of a DSL inside another language has both advantages and disadvantages. A major advantage is the reuse of the host language's syntax and semantics. For example, if a DSL requires expressions or interacting with external libraries, support for this can be automatically inherited from the host language. This can also be a disadvantage, since it is impossible to prevent the developer of a program in the DSL from using the host language to step outside its intended scope.

An *external DSL* is another approach to DSL implementation. In this case, the DSL is entirely separate from any other language or tool, providing the designer with greater freedom to design its syntax and determine what a user can do with it. Comparable to general-purpose programming languages, two common implementation approaches to external DSLs exist: interpretation and compilation.

In both cases the DSL has its own concrete syntax that exists separately from the interpreter or compiler that transforms it, either to a runtime representation (interpreted) or an output format that is either directly executed or input to a lower level transformation tool (compiled). In both situations, from an implementation perspective, they are the same as an interpreter or compiler for a general-purpose programming language.

Hybrid approaches exist as well. For example, a general-purpose programming language can be extended with domain-specific syntax that can be automatically mapped to code in the host language [BV04, ERKO11]. While this approach allows restricted reuse of the host language, it does require the maintenance of a general-purpose programming language extension.

1.3 Towards Model-Driven Digital Forensics

A key concern in all phases of automated digital forensics is handling *variability*, since there is no control whatsoever over the input: any digital device may contain relevant information. We have selected the recovery phase as the focus of our research since we believe it will benefit the most from a model-driven software engineering approach, since it is performed entirely in software.

Acquisition heavily relies on custom, manual approaches, including hardware repair and replacement along with the use of hardware devices such as write blockers and faraday cages. In general it is related to the relatively low pace of change in hardware used in practice. The resulting challenges therefore are mostly outside the scope of a software engineering approach such as MDSE.

Analysis will likely benefit from model-driven software engineering approaches in the future. However, since digital forensics-specific analysis techniques are

mostly in their infancy, it is currently not possible to accurately determine the domain's coverage and concepts, which are key requirements in order to apply MDSE.

There is currently a very small amount of literature on specific digital forensics analysis techniques and forward-looking discussions mention it as an area that requires substantial research and development effort [Bee09, Gar10].

Variability in Recovery

Once acquisition of data during an investigation is complete, a lot of variability remains. All digital data consists of bits, but how those bits are organized in order to form information can differ greatly. Network dumps consist of interlaced streams of packets or messages. Different levels of protocols encapsulate each other's data and in the process sometimes concatenate or truncate it.

Digital storage devices were traditionally disk-based devices that used sectors as smallest possible units of storage. Usage of these devices was optimized around the sectors that were quickest to access. Currently solid-state drives (SSDs) are gaining in popularity. SSDs are laid out differently, related to the process of *wear leveling* [LNTG05], which leads to spreading out data across the entire device evenly in order to increase its longevity.

These variations lead to different recovery techniques in order to reconstruct and interpret the acquired data. Still, once an approach to efficiently exchange messages, optimize for storage in the fastest sectors or spread out data on an SSD reaches a certain level of efficiency, the industry tends to standardize around it. The recovery software that handles it then requires relatively little maintenance.

The farther we move away from the lowest level of data storage, the more variability we encounter, as each level makes it progressively easier to build different abstractions on top of the previous one. This creates an inverted pyramid, as depicted in Figure 1.3. While this image illustrates the increasing variability with regard to digital storage devices, the view is similar for networking. In fact, the OSI model [Zim80] is often displayed in a similar manner.

At the lowest level, all digital data is organized into strings of values that are either 1 or 0. They are stored in a limited set of hardware storage device types, such as hard disks and memory chips. To manage their storage performance and reliability characteristics, a bigger set of hardware/software solutions exist, including RAID. On top of that, operating systems support a growing set of file systems, such as NTFS. These file systems then store the actual files, within which two levels can be distinguished, of files that are actually small portable file systems such as ZIP, and regular top-level files such as JPEG[2].

[2]Which itself may contain additional files in different formats, such as thumbnails.

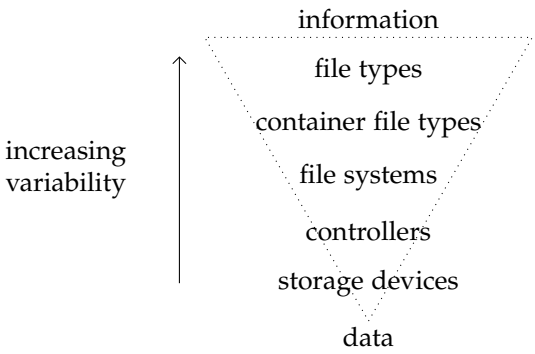

Figure 1.3: The inverted pyramid of variability in storage abstractions.

This means that the biggest challenge in dealing with variability lies at the level of top-level application file formats. Apart from this resulting from our analysis, it is also our intuition after more than a decade of engineering automated digital forensics tools. This variability is caused by a multitude of factors, that amplify each other:

Operating Systems: Different operating systems have different file formats, including logs, configuration files and caches.

Applications: Individual applications such as web browsers and mobile apps include their own logs, caches and file types (e.g., documents and cookies).

Versions: File formats tend to be revised regularly to support new capabilities, but in practice each version of a format will be encountered.

Intended Variants: Vendors sometimes extend an existing or even standardized format to support capabilities unique to their devices or applications.

Unintended Variants: Popular formats are produced by a large amount of different applications, some of which contain bugs in their serialization code.

Engineering Recovery Tools

There is considerable activity in the engineering of digital forensics recovery tools. Some tools are focused on specific domains, such as networking [AT05] or embedded devices [BK05]. Others are specifically aimed at recovering data from multiple sources, either by implementing or aggregating different tools [BBB+12], or by ignoring the differences in order to improve runtime performance and scalability [Gar13].

Most modern tools adhere to good design principles in their implementation[3] such as separation of concerns. However, they are implemented in general-purpose languages and as a result, this separation will never be perfect since some concerns cross-cut others [TOHSJ99].

Our approach to move the development of recovery tools to model-driven software engineering is twofold. First, we have developed a DSL to lift the specification of file formats to a higher level of abstraction, in order to improve productivity in dealing with variability. Second, we have decoupled the implementation of the code handling these file formats from their (evolving) specification, so that other non-functional requirements such as high runtime performance and scalability can be realized independently from the file format specifications.

The need for investigation of this direction is the result of previous research and development efforts, including experience with tools such as Xiraf [BBB+12] and Defraser [Net05]. These tools have shown the usefulness of extensive automation in the domain of digital forensics, but have also identified the need for effective solutions to deal with large amounts of variability, especially in the area of file formats and protocols.

1.4 Research Questions and Perspectives

This thesis is concerned with the study of applying model-driven software engineering in the domain of automated digital forensics. More specific, the design, development and evaluation of a domain-specific language to allow the specification and maintenance of forensically relevant file formats. Additionally, the design, development and evaluation of an accompanying implementation that optimizes for the other non-functional requirements: high runtime performance and scalability. This section discusses the research questions that were investigated and the relevant research perspectives.

Main Research Question:

> *Can we improve the practice of engineering automated digital forensics tools through the application of model-driven software engineering techniques, specifically in the domain of recovering information stored in files?*

In order to specify what is meant by the word *improve*, we break this question down into several questions, related to the functional and non-functional requirements for automated digital forensic recovery tools.

[3] Based on an assessment of open source file carvers discussed in Chapter 3

Q1: Can we separate the concerns in file format specification from their implementation?

Q2: Can we determine what the runtime performance costs are of separating the concerns of file format specification from their implementation?

Q3: Can we leverage model transformation to tune the scalability and runtime performance of our solution?

Q4: Can we determine whether our solution provides the modifiability required in practice?

We address Q1 in Chapters 2 and 3, by performing domain analyses that lead to Derric, a DSL to declaratively describe file formats. Q2 is addressed in Chapters 3 and 4, by evaluating the data recovery tool Excavator that uses Derric descriptions on standard and custom benchmarks.

In Chapter 4 we address Q3 through the development and use of a set of optimizing model transformations and a custom 1TB benchmark. Finally, Q4 is first addressed by performing a set of realistic maintenance scenarios on Derric descriptions, which is discussed in Chapter 5. In relation to those maintenance activities we additionally discuss Trinity, a supporting IDE to simplify the debugging of Derric descriptions, in Chapter 6.

Research Perspectives

Several perspectives can be applied to the research presented in this thesis, all related to the application of model-driven software engineering. The first is about MDSE in general, the other two about applying specific technologies:

Model-Driven Software Engineering in Practice: We present data on the feasibility and practical applicability of model-driven software engineering.

Derric: Applying MDSE in Automated Digital Forensics: We evaluate tools, including Derric and Trinity, in order to determine benefits and drawbacks of using MDSE in the domain of automated digital forensics.

Rascal: DSL Engineering in Practice All MDSE-specific automation is implemented using Rascal, presenting observations on its use in the domain in realistic scenarios.

1.5 Software and Technology

This thesis describes an evaluation of model-driven software engineering in practice. As such, a considerable part of the total effort concerned the development of software used in the experiments.

All software is open source[4] and consists of the following major components:

Derric: A DSL to describe file formats. Its implementation consists of:

- **Compiler front-end:** Implemented in Rascal, includes grammar, optimizations and a custom intermediate (platform-independent) language.
- **Code generator:** Implemented in Rascal, generates Java source code implementing file format validators.
- **Interpreter:** Implemented in Java, executes the file format validator implemented in the front-end's intermediate language.

Excavator: A file carver. Implemented in Java, interfaces with components created by the Derric code generator.

Trinity: An IDE for Derric. Implemented in Java, interfaces with the Derric compiler front-end and interpreter.

Utilities: Several tools to automate research tasks:

- **Fraggen:** Hard drive image generator with support for fragmented files.
- **FileHerder:** Runs Derric generated code on sets of files and collects results.

To give an impression of the size of the developed software, Table 1.1 shows the non-empty lines of code, along with the chapters the software is used in.

Component	Rascal	Java	Chapter
Derric	2.346	2.041	3,4,5,6
Excavator		1.416	3,4
Trinity		1.009	6
Utilities		579	4,5
Total	2.346	5.045	

Table 1.1: Component sizes and relevant chapters.

[4] Available from: `http://www.cwi.nl/model-driven-engineering-in-digital-forensics`.

1.6 Origin of Chapters

Chapter 2. Towards an Engineering Approach to File Carver Construction.

Accepted at The Third IEEE International Workshop on Computer Forensics in Software Engineering (CFSE'11). Published in the proceedings of the *35th Annual IEEE Computer and Software Applications Conference Workshops (COMPSACW'11)* [AB11]. Joint work with Leon Aronson.

Chapter 3. Bringing Domain-Specific Languages to Digital Forensics.

Published in the proceedings of the *33rd International Conference on Software Engineering (ICSE'11)* [BS11]. Joint work with Tijs van der Storm.

Chapter 4. Domain-Specific Optimization in Digital Forensics.

Published in the proceedings of the *5th International Conference on Model Transformation (ICMT'12)* [BS12]. Joint work with Tijs van der Storm.

Chapter 5. A Case Study in Evidence-Based DSL Evolution.

Published in the proceedings of the *9th European Conference on Modelling Foundations and Applications (ECMFA'13)* [BS13a]. Joint work with Tijs van der Storm.

Chapter 6. Trinity: An IDE for The Matrix.

Published as a tool paper in the proceedings of the *29th IEEE International Conference on Software Maintenance (ICSM'13)* [BS13b]. Joint work with Tijs van der Storm.

CHAPTER 2

Towards an Engineering Approach to File Carver Construction

This chapter was previously published as a paper with the same title in *35th Annual IEEE Computer and Software Applications Conference Workshops (COMPSACW'11)* [AB11]. Joint work with Leon Aronson.

Abstract

File carving is the process of recovering files without the help of (file system) storage metadata. A host of techniques exist to perform file carving, often used in several tools in varying combinations and implementations. This makes it difficult to determine what tool to use in specific investigations or when recovering files in a specific file format.

We define recoverability as the set of software requirements for a file carver to recover files in a specified file format. This set can be used to evaluate what tool to use or which technique to implement, based on factors such as file format to recover, available time, engineering capacity and data set characteristics.

File carving techniques are divided into two groups, format validation and file reconstruction. These groups refer to different parts of a file carver's implementation. Additionally, some techniques may be emphasized or omitted not only because of file format support for them, but based on performance effects that may result from applying them.

We discuss a variant of the GIF image file format as an example and show how a structured analysis of the format leads to design decisions for a file carver.

2.1 Introduction

Forensic data recovery tools such as *file carvers* [PM09] use a large amount of different techniques to recognize (parts of) files, ranging from simple recognition approaches such as magic number matching [RR05] to more advanced and involved approaches such as file structure validation [Gar07] and statistical fragment classification [KS06] [Vee07].

Research in data recovery techniques typically provides empirical results of effectiveness on different types of files as well as a discussion of the underlying causes. However, it is difficult to properly evaluate the complexity of recovering a file of a newly introduced file format or the expected effectiveness of a proposed recovery technique. Having this capability would make it easier to decide what techniques to use during digital forensics investigations as well as assist developers creating new file formats to make design decisions that improve a file format's *recoverability*.

We propose a definition of recoverability specific to digital forensics based on the software engineering requirements of implementing a file carver for a file format. These requirements consist of techniques that are a reflection of two factors that influence recoverability. First, the difficulty of automatically validating that any given block of data conforms to the given file format. Second, the impact that outside effects, such as fragmentation, have on the difficulty of recovering a file of the given file format.

These requirements can be used in several ways. First, to determine the complexity of (automatically) constructing a file carver for a given file format. Second, to help decide what file formats to search for or which file carver (or algorithm) to use during a digital forensics investigation. Finally, to guide application developers in creating file formats that are relatively easy to recover.

This chapter is organized as follows. Section 2.2 discusses the techniques that are available as requirements for a file carver for a given file format. As such it also serves as a discussion of related work in the domain of file carving. Section 2.3 discusses the impact these techniques have on the performance of file carvers and the recoverability of the file formats they support. Section 2.4 presents an example discussing all the issues presented before and demonstrating their use. Section 2.5 discusses the suitability and applicability of the proposed approach. Section 2.6 concludes.

2.2 File Carving Techniques

If metadata describing where a file is stored (usually as part of a file system) is missing or inaccessible, a content-based approach can be used to recognize and reassemble available data in an attempt to recover files. *File carving* is the term used

for all combined approaches in this area. Two factors influence how difficult it is to recover a file using file carving: the file's own format as well as the state of the (surrounding) data. The following subsections discuss these two factors and split them into separate techniques that can be implemented independently.

Format Validation

The easiest way to determine whether a block of data conforms to a given format, is to load it into an application accepting that format and, if it loads, manually inspect the loaded file to see if the content makes sense. However, when recovering data this approach is generally unfeasible: from terabytes of data, millions of files can potentially be recovered, which could take months to inspect manually.

Cutting out user intervention increases feasibility considerably, which is achieved by using an automated *format validator*. This is a program (or function in a system) that accepts a block of data and determines whether it conforms to the defined structure of the file format it validates.

Although this approach is typically orders of magnitude faster than manual format validation, major scalability issues remain. The more strict a validation is, the more computing power it generally requires. For example, validating compressed files may require decompressing all contents and calculating multiple hashes over large amounts of data.

Whether automated format validation is at all feasible also depends on the file format's defined structure. If the structure is only loosely defined and does not have any internal verification mechanisms (such as using length fields or an embedded hash) it may even be impossible to automate format validation.

Several approaches exist to perform automated format validation, all related to aspects of existing file formats. Following is a discussion of those approaches, ordered by increasing complexity.

Magic Number Matching

Binary file formats typically use *magic numbers*, identifiers that signal the beginning (or end) of a file or internal data structure. For example, GIF files always start with the ASCII string `"GIF"` and end with the byte `0x3B`. A validator using magic numbers only needs to compare values in order to make decisions. However, except for the bytes containing the magic numbers, not much is known about the data.

Scalpel [RR05], the successor to ForeMost, is one of the most popular file carvers and nearly exclusively uses this technique.

Data Dependency Resolving

The use of *data dependencies* allows file formats to parameterize (parts of) their own layout. For example, BMP files contain length fields that specify both the size of the entire file and of an internal data structure, as well as a flag specifying whether a color table is embedded in the file. Interpreting these values can help validators to locate possible inconsistencies (e.g., when the end of a block described by a length field is not followed by the next expected data structure) but even then, the actual contents of the data blocks are not validated.

Internal Verification Checking

As opposed to the previous two approaches, *internal verification* does take the actual contents of (part of) a file into account. For example, PNG files consist of a series of so-called chunks, which are blocks of data that specify their length, type (using magic numbers), contents and a cyclic redundancy code (CRC) over the type and contents.

While calculating CRCs takes time, a validated block of data is very likely to be correct. Configuring which verifiable parts should actually be verified can be used to reduce the required time.

Algorithm Output Analysis

Most file formats that are interesting in a digital forensics context employ some kind of encoding or compression. *Output analysis* examines the encoded or compressed data as stored in the file in relation to the algorithm that was used to create it. For example, JPEG uses Huffman coding to compress data. Given a block of data, it is possible to determine whether it was likely compressed with a given Huffman table using bit sequence matching [SM09].

Compressed/Encrypted Data Decoding

Data decoding as part of validation is basically an automated version of the manual validation process without the inspection. For example, successfully decoding an MPEG file has a high chance of yielding at least a partially viewable movie. Additionally, if the decompression is only partially successful, the location of the error is usually close to where the corrupted or missing data is. However, it does require significant computation, even compared to typical internal verification or even output analysis.

File Reconstruction

If all data were stored in single consecutive blocks and never overwritten, data recovery would be nothing more than running all available format validators on a block of data and collecting the resulting files. In practice, operating systems implement a host of performance optimizations that both enable and complicate data recovery.

The biggest performance gain for file systems is typically achieved by not actually removing files upon deletion, but simply marking their location as available for writing. This optimization makes file carving at all possible. File fragmentation is an optimization that causes files to be split into several parts and scattered over the physical contents of a storage device. This complicates data recovery significantly.

Without metadata, it is difficult to determine the original order the fragments were stored in. Attempting all possible combinations of a set of fragments is intractable. *File reconstruction* is concerned with employing heuristics such as knowledge of typical fragmentation patterns or file characteristics in order to reduce the search space. Following is a discussion of the approaches in this area.

Fragment Reordering

File reconstruction based on *fragment reordering* attempts a subset of all possible combinations of available fragments and uses a set of format validators to determine matches. There are two general approaches to compute this subset and keep implementations within an acceptable running time.

Bifragment Gap Carving [Gar07] restricts the search space by only carving fragmented files that are split into two fragments that occur consecutively on the physical storage. First, a block of data starting with a header and ending with a footer that is rejected by the format validator is located. All possible combinations of fragments that make up this block are then attempted, under the constraints that no fragments are reordered and that all removed fragments are contiguous. In effect, all embedded "gaps" are attempted.

Advanced Carving [Coh07] uses a format validator not only to accept or reject files, but also to determine the location in rejected files where the fragmentation has occured. As a result, only certain file formats (that have extensively defined internal structure) and format validators (that implement data dependency resolving or data decoding) can work with this approach. It can recover files where the fragments are out-of-order on the physical storage, but on realistic data sets is only tractable when recovering files split into two fragments.

Fragment Classification

An alternative approach to reducing the amount of possible combinations of fragments to consider when reconstructing files is *fragment classification*. Individual fragments are considered and, based on their contents, either included or excluded from further reconstruction. As such this approach combines well with fragment reordering or any other file carving technique, as it simply reduces the amount of fragments to consider.

Classifiers are generally implemented in the form of supervised learning applications using some metric that helps recognize different types of file fragments. Experiments in this area have been conducted using a diverse set of metrics, including byte frequency analysis and byte frequency correlation analysis [MH03], Shannon entropy, chi-square distribution and Hamming weight [CBS+10] and Normalised Compression Distance [Axe10].

Although classification techniques all have their own characteristics, a general observation is that compressed and encrypted data is easy to recognize, but hard to classify. Data that has not been encoded, such as plain text or bitmap files are generally easy to classify, because they have easily identifiable characteristics (e.g., plain text only uses a subset of all byte values and bitmaps often have distinct patterns such as having a zero every four bytes as alpha channel value).

2.3 File Carving Performance

The file carving techniques discussed in the previous section all enable automated file carving and have some performance benefit or cost for a file carver that employs them. In general, without a limitation on the amount of combinations of fragments considered, no matter how fast a format validator is, the resulting running time on an average hard drive can be months or years. At the same time, format validators may require significant computation to come to a conclusion. These two factors are discussed in the following subsections.

Format Validator Invocation Reduction

Recovering fragmented files is a combinatorial problem: all combinations of fragments in the set of the smallest unit of data on a data storage device are to be attempted to discover files that originally resided on the device. An attempt in this context is an invocation of the format validator to determine whether a match was found. This solution is intractable even when the large amounts of data involved in practice are ignored. Two approaches are used to reduce the amount of times the format validator is invoked.

The first is to only consider a subset of all fragment combinations, which is what all practical fragment reordering techniques such as bifragment gapcarving do. The algorithm simply only looks for files that have been fragmented in a certain manner that is common in practice, for example, fragmentation into two parts [Gar07].

The second is to use the results of each format validator invocation or some other program or function to reduce the data set either by eliminating or grouping fragments. Elimination is achieved using techniques such as magic numbers, by excluding all fragments that do not start with some fixed value, and fragment classification, by excluding all fragments that do not match the statistical properties of the file format that is being recovered. Grouping is achieved by all format validation techniques that support partial validation by grouping fragments together once they partially validate (typically the start of a block up to a certain point) in some attempted order. The grouped fragments can then be considered a single (larger) fragment, reducing the size of the data set.

Format Validator Computation Reduction

The amount of data on current data storage devices is growing to such a size that reducing the amount of fragment combinations to consider from the original intractable solution to a polynomial-time solution may still require days or weeks of processing time dependent on the performance of the format validator.

For example, when considering a relatively small block of 100MB of data consisting of typical 512-byte sectors, a quadratic function to determine all possible candidates for validation requires billions of format validator invocations. This makes it extremely important for format validators to only perform computations that are crucial to validate files in a given file format.

One possible approach is to not implement computationally expensive validation techniques such as output analysis and data decoding and accept a small amount of false positives, especially given that eventual evidence will have to be manually inspected. If the percentage of false positives is manageable in this manner, it may pay off to accept them and handle them in the manual stage.

2.4 Recoverability Example: GIF

Based on the techniques enumerated in Section 2.2 and the performance considerations discussed in Section 2.3 it is possible to assess what combination of techniques can either be discarded or included in the software engineering requirements for a file carver for a given file format, as well as how to use those techniques effectively. As a practical illustration, we discuss a simplified structure of the GIF image file format.

```
format gif
extension gif

strings ascii
sign false
unit byte
size 1
type integer

sequence
Header
([Image ComprBlk* ZeroBlk]
 [AppExt DataBlk* ZeroBlk])*
Trailer

structures
Header {
    Signature: "GIF";
    Version: "87a" | "89a";
    LSWidth: size 2;
    LSHeight: size 2;
    Flag: unit bit;
    ColorResolution: unit bit size 3;
    SortFlag: unit bit;
    Size: unit bit size 3;
    BGColorIndex;
    PixelAspectRatio;
    GCT: size Flag*(3*(2^(Size+1)));
}

Image {
    Separator: 0x2c;
    Left: size 2;
    Top: size 2;
    Width: size 2;
    Height: size 2;
    Flag: unit bit;
    InterlaceFlag: unit bit;
    SortFlag: unit bit;
    Reserved: unit bit size 2;
    Size: unit bit size 3;
    LCT: size Flag*(3*(2^(Size+1)));
    LZWMCS;
}

AppExt {
    ExtensionIntroducer: 0x21;
    ExtensionLabel: 0xff;
    BlockSize: 11;
    AppId: type string size 8;
    AppAuthCode: size 3;
}

DataBlk {
    Length: 1..255;
    Data: size Length;
}

ComprBlk = DataBlk {
    Data: lzw(packing=lsbfirst,
              codesize=variable,
              startsize=Image.LZWMCS)
          size Length;
}

ZeroBlk = DataBlk {
    Length: 0;
}

Trailer { Marker: 0x3b; }
```

Figure 2.1: Simplified structure of the GIF image file format.

A description of the structure we discuss is shown in Figure 2.1. The structure is expressed in Derric (see Chapter 3), a digital forensics-specific data description language that we have developed to precisely express the structure of data formats in order to allow extensive analysis.

Simplified GIF Format

A simplified structure of the GIF image file format discussed is shown in Figure 2.1. The only simplification that has been applied is the exclusion of some extension structures due to size constraints in this chapter. As a consequence, a file that

adheres to this specification is a well-formed GIF image file, making this example realistic.

The specification identifies the name of the format (line 1) and its file extension (line 2) along with a set of defaults: strings use the ASCII character set (line 4) and whenever the specification of binary values is omitted, they are unsigned, single-byte integers (lines 5-8). The rest of the specification is divided between the specification of the file format's sequence (lines 10-14) and structures (lines 16-70).

The terms used in the sequence section refer to defined structures of the same name in the structures section. Additional characters are used to define grammatical aspects of the file format, such as optionality (question mark), repetition (asterisk), alternatives (parentheses) and fixed order subsequences (square brackets).

As a result, the defined sequence prescribes that every GIF file starts with a `Header` and ends with a `Trailer`. In between is an arbitrary amount of any combination of two subsequences: starting with an `Image`, followed by any number of `ComprBlk`s and terminated by a `ZeroBlk` or starting with an `AppExt`, followed by any number of `DataBlk`s and terminated by a `ZeroBlk`.

The structures referenced in the sequence are defined in the structures section. Every structure has a name along with a list of its fields. Each field has a name and a specification of its contents. Every part of the specification that is not defined is based on the defaults specified at the top of the description (lines 4-8). For example, the `Size` field in the `Image` structure (line 41) has an unknown value (not specified), but its type is an unsigned integer (not specified, based on defaults) with a size of 3 bits (specified).

GIF File Carving

Developing a custom file carver for the simplified GIF image file format requires an analysis of its specification and a definition of which techniques in Section 2.2 can be used to maximize the amount of recovered data, without using intractable approaches that will often run for months in practice, as discussed in Section 2.3.

GIF Format Validation

Every GIF file starts with a fixed header (line 18) and terminates with a fixed trailer (line 70), enabling the use of *magic number matching* to find complete files.

data dependency resolving presents an interesting addition as the format contains several flags to signal the existence of other data structures (lines 22 and 37) and mandates length fields on all `DataBlk`(-based) structures (lines 54-64), including a prescribed `ZeroBlk` terminator (lines 66-68).

GIF files do not contain mechanisms for *internal verification checking*, but *algorithm output analysis* can be performed, especially given the well-known compression algorithm (LZW) and variable starting size for code tokens (lines 60-62). Apart from analyzing the tokens, *compressed data decoding* can be used to fully validate the compressed data stream.

GIF File Reconstruction

Extensive structure in the GIF format based around small length-specified `DataBlk`s and a well-known compression algorithm make it relatively easy to develop a format validator that is capable of fairly precisely pinpointing fragment boundaries when attempting to reconstruct a fragmented file, so *Fragment Reordering* approaches such as bifragment gapcarving can be applied.

Applying *fragment classification* however is more difficult. While the compressed data will be relatively easy to recognize, the GIF image file format also allows `AppExt` structures that may contain data that is not compressed, such as plain text comments or even embedded text that is part of the image. So while classification can be useful to identify possible fragments that may be part of a fragmented GIF file, it is not recommended to discard fragments based on not being classified as compressed.

GIF File Carving Performance

GIF files tend to be of limited size because larger (photographic) images are often stored as JPEG or PNG, since they support more colors and better compression. The smaller files typically are, the less they tend to be fragmented. Combined with the opportunities in the file format to create a format validator that can fairly precisely pinpoint fragment boundaries, the amount of format validator invocations will be small compared to other media formats.

However, to maximize the amount of files that can be recovered, the *algorithm output analysis* and *compressed data decoding* may be omitted completely. Instead, format validation can rely on *data dependency resolving* fully. This is possible because each `DataBlk`(-derived) structure must specify its single byte length up front, allowing easy detection of errors at a very high granularity of 256 bytes, half the size of typical 512-byte sectors on storage media. Only checking length fields will significantly reduce the amount of computation a format validator has to perform.

2.5 Discussion

Given the diversity in file formats, file carving techniques, performance considerations and data storage systems, there are several cases where our approach to

documenting the recoverability of files based on their format's enabled file carving techniques raises questions about applicability and suitability. Following is a discussion of the questions that we have currently identified.

Outside Factors

There are several factors that impact what the actual contents of files in a given file format is made up of. The first is related to the applications that generate the files. All kinds of design decisions were made by the developers of these applications that impact the recoverability of the files the applications create. For example, whether or not to use the optional restart markers in JPEG files, or whether to split the compressed data of a PNG file into separate IDAT structures. Both would make it much easier for file carvers to recover those files, but since they both rely on implementation aspects related to optional features in the file format, they are difficult to integrate into an objective model.

The second factor deals with actual contents of the files. Bitmap files that contain uncompressed data and that don't use the alpha channel are easy to classify, but it is unclear what that means for the entire format. Another example is file size growth. With photo and video cameras producing larger and larger files, the ratio between metadata and (compressed) contents is constantly changing, which may have an impact on how difficult it is to carve files of that type.

Technique Selection

There is no fixed process describing how to proceed after enumerating the types of file carving techniques that can be used to recover files of a given file format. In general, it is important to have a validator that maximizes precision, because more than 90% of files are not fragmented [Gar07] and can be recovered without requiring any type of file reconstruction. Beyond that it is difficult to decide on what technique to implement first, especially since the specific encoding or compression algorithms of the different file formats greatly impact the difficulty of implementing the more advanced format validation techniques, such as *algorithm output analysis*.

Still, a structured assessment of the file carving techniques enabled by features of specific file formats does lead to insights about how or if to apply them. An example is in our discussion of the GIF image file format. Without careful analysis, it may appear obvious that such a compressed format will require *compressed data decoding* or *fragment classification*, while in practice, the extensive use of a small length field makes faster approaches also feasible.

Engineering Effort

Digital forensics investigations are often performed under high time pressure due to deadlines related to legal proceedings such as pre-charge detainment. As a result, apart from precision and performance, another factor is present when dealing with files of a file format that was previously unsupported: engineering effort. When deciding what technique to implement, these three factors must be taken into account. For example, if a file format's embedded data supports high-speed decompression and high precision with regard to locating corrupted data, implementing *compressed data decoding* for this file format will probably result in fast file carving. However, if implementing support for this feature takes up a large amount of time, it might be more efficient to implement slower and less precise techniques in the validator so that the file carving may start earlier.

Format Engineering Implications

Using file carving techniques to develop easily recoverable file formats may result in some unusual design decisions. For example, while it is generally considered good practice to implement existing standards instead of inventing or modifying a new compression algorithm, custom data formats tend to make file carving easier. For example, the escaping in JPEG makes *algorithm output analysis* possible. However, we believe it is a useful tool to assess the recoverability of files during file format development, since most of the practices it promotes are in line with general engineering guidelines.

Following the techniques described in this chapter, a file format developer would be advised to use:

- Magic numbers for at least header and trailer.
- Length fields for all data structures.
- Flags to indicate the existence of all optional data structures.
- Checksums to protect the integrity of all data structures.
- Encoded data only if the application requires it.
- Small data structures that will require minimal fragment reordering to recover.

2.6 Conclusion

File carving has been the subject of active research for the past decade and has resulted in two techniques that we have discussed in this chapter: *format validation*

and *file reconstruction*. The first focuses on validating that a block of data adheres to a given file format and file reconstruction focuses on reassembling fragmented files. In practice both techniques are combined so that file carving can be almost entirely automated.

However, automation can easily result in a solution that will still be unfeasible due to the large amount of time required to carve a single file. An important conclusion is that a file carver that simply implements all file carving techniques that a file format supports may not be an optimal solution in practice.

In this chapter we have proposed to take all aspects of a file's format into account and consider each technique within the context of accuracy and performance. This will lead to design decisions that are both precise with regard to reducing false positives as well as scalable with regard to recovering data in an acceptable running time.

To illustrate our approach, we have presented an analysis of a simplified GIF image file format and show that considering each technique and combination can lead to different design decisions.

Future work

We are currently developing an automated file carving framework based on model-driven engineering (see Chapter 3, including methods to automatically infer the file carving techniques supported by a given file format in order to generate components for the framework. This would for example enable a user to enter a time limit, which is then used by the application to select a suitable set of carving techniques, thus optimizing the results in a given time frame.

Part II

Modularity and Efficiency

Highlights:

- Design and implementation of the DERRIC DSL and compiler.
- Implementation of the EXCAVATOR data recovery framework.
- Implementation of model transformations to tune runtime performance and scalability.
- Development of custom 1TB data recovery benchmark to exercise DERRIC.
- Benchmarking of EXCAVATOR/DERRIC on existing and custom benchmarks.

CHAPTER 3

Bringing Domain-Specific Languages to Digital Forensics

This chapter was previously published as a paper with the same title in *33rd International Conference on Software Engineering (ICSE'11)* [BS11]. Joint work with Tijs van der Storm.

Abstract

Digital forensics investigations often consist of analyzing large quantities of data. The software tools used for analyzing such data are constantly evolving to cope with a multiplicity of versions and variants of data formats. This process of customization is time consuming and error prone.

To improve this situation we present Derric, a domain-specific language (DSL) for declaratively specifying data structures. This way, the specification of structure is separated from data processing. The resulting architecture encourages customization and facilitates reuse. It enables faster development through a division of labour between investigators and software engineers.

We have performed an initial evaluation of Derric by constructing a data recovery tool. This so-called *carver* has been automatically derived from a declarative description of the structure of JPEG files. We compare it to existing carvers, and show it to be in the same league both with respect to recovered evidence, and runtime performance.

3.1 Introduction

Digital forensics is the branch of forensic science where information stored on digital devices is recovered and analysed in order to answer legal questions. The continuous growth of storage size and network bandwidth and the increased popularity of digital hand-held devices, makes digital forensics investigations increasingly dependent on highly customized data analysis tools. Only the use of extensive automation offers a means to deal with the scale of current and future investigations. Apart from raw scale, the diversity in types of devices, storage and memory layouts, protocols and file formats requires an equally impressive flexibility in these tools: in order to deal with emerging and changing data formats they must be continuously evolved, customized, and redeployed.

Data formats are often poorly documented and hence must be reverse engineered. Even if data formats are documented, there are often many variants that require changes to the implementation of the data format processor. Additionally, off-the-shelf data format processors such as spreadsheets or image viewers are often inadequate, since in digital forensics, one often has to deal with incomplete or otherwise corrupted data: such fragments may contain crucial evidence.

The challenge for software engineering in digital forensics is therefore:

> **How to construct high-quality data analysis tools that are easy to modify and customize, and yet at the same time are able to handle data in the terabyte range?**

To achieve both the required scalability and flexibility we propose an architecture that separates the development of the data analysis tools from the data format processors. This allows the data analysis tool to be optimized for maximum scalability and define how data format processors must be implemented to be usable in the tool. Additionally, data format processors are developed using a data description language that allows declarative specification of data formats. These specifications are then transformed using a code generator into the form the data analysis tool requires. This approach simplifies development (by separating data formats from processing algorithms) and allows for optimizations (by the code generator, either based on data analysis tool requirements or opportunities in the data formats).

For data description, we propose a domain-specific language called Derric that is designed to accomodate the workflow of a digital forensics investigator, implementing constructs that match typical activities such as reverse engineering, iterative development and using data format documentation. To evaluate our language, we describe its use in a typical forensics scenario. Additionally, to evaluate our entire architecture, we develop an instance of our system implementing a typical digital forensics data analysis tool doing file carving, the process of recovering

deleted, hidden or obfuscated files from a data storage device. We compare our tool to existing relevant file carvers and show that our system performs as good as industrial-strength carvers while being much more flexible.

This chapter makes the following contributions:

- An analysis of the software engineering challenges in digital forensics, mapped to practical issues.
- The digital forensics-specific data description language Derric.
- An evaluation of a Derric-based data analysis application in comparison to industrial-strength tools on standard benchmarks.

Organization of this chapter

The rest of this chapter is organized as follows. Section 3.2 discusses the software engineering challenges in digital forensics and maps them to practical problems in data analysis tools. Section 3.3 presents the data description language Derric, demonstrating its use in a typical scenario. Section 3.4 presents an instance of our complete architecture in the form of a file carving tool utilizing Derric. The evaluation comparing our system to existing file carvers is also presented. Section 5.7 discusses issues around suitability and applicability of our work. Section 3.6 discusses related work both in the area of domain-specific and data description languages as well as in digital forensics. Section 5.8 concludes.

3.2 Digital Forensics Challenges

The most important challenges in digital forensics include domain-specific data abstraction, modularization and improving scalability [Gar10]. Data abstraction deals with the need for one or several standard formats to describe, store and use data in different formats. Modularization refers to the need to increase and deepen integration between digital forensics tools to reduce manual preparations and allow extensive reuse between types of tools (e.g., using the same tool to recover images from both a storage device and a network stream). Scalability is important to keep digital forensics investigations feasible in the face of current and future storage capacities, bandwidth and device use. Below we discuss the domain-specific aspects of these challenges in more detail.

Data Abstraction

Digital forensics investigations typically require the support of a large amount of data formats, ranging from file systems and formats to protocols and memory lay-

outs, where each class can have several different instances depending on type, version and implementation. An example of this is the FAT file system, which has multiple types: FAT12, FAT16 and FAT32, where FAT16 has two versions. All types and versions are implemented by multiple operating systems.

Reverse Engineering

Whenever data is encountered in an unknown (or known, but proprietary) format, a process of reverse engineering starts to recover enough of the format's structure to be able to recover files of this type or extract information from recovered files.

A common problem is the distance between the encoding of identifiers discovered in the data under investigation and the format in which they must be expressed. If the notation doesn't support the same encoding, the data must first be transformed. Besides being error prone, it also obfuscates the description. Examples of different encodings are string encodings such as ASCII and unicode and numerical values of any bit size.

Fragmented or missing data is also common. If a data description method does not support the expression of parts of the data that are currently unknown, the rest of the format can either not be expressed or the unknown data must be described using some approximation. This prevents the tool using the description from using the knowledge of missing data to its advantage by choosing a method appropriate to its requirements and capabilities. Additionally, expressing what parts of a format are currently unknown instead of some arbitrary placeholder increases its value as documentation.

Using Documentation

In another situation, the format of the data that is encountered in an investigation is well-known and documented. In this case the documentation is used to create an implementation of the format in order to recover files of this type or extract information from recovered files.

A similar problem occurs here regarding the distance between the encoding of identifiers in the documentation and the format in which they must be expressed. Although data format documentation tends to map relatively cleanly to implementations in data description or programming languages, some important exceptions exist. The most common is in the formatting of strings, where data formats typically still use ASCII strings, the default format for strings in programming languages has typically evolved to a type of unicode, or is dependent on external factors such as compiler options, linked libraries or runtime platform. This functionality typically exists so applications can easily be adapted but may have unwanted consequences in a data format processor.

Another problem related to the encoding issues is that documentation may present data in a different format on purpose. An example is the Microsoft Office file formats documentation [Mico8], that displays all bit diagrams in big-endian byte order for readability even though it requires implementations to store the actual files in little-endian byte order.

Iterative Development

Regardless of the approach used to develop the data format description, the process is typically highly iterative for several reasons. The smallest possible description that will reliably lead to recovering evidence is always sought since strict deadlines are common. To find this description, it is developed iteratively, checking at every increment whether it succeeds. This effectively requires the process to go from describing to executing to be simple and fast. In an ideal situation, this means that data analysis tools can be reconfigured or extended at runtime, or have capabilties to easily and quickly shutdown and restart.

Modularization

The diversity in types of digital forensics investigations is high, ranging from the analysis of a regular confiscated data storage device, such as a hard drive, to a highly specialized embedded device such as a detonator. At the same time, as reuse of techniques and formats between devices is high, the reusability of the tools analyzing them should be as well. An example is a file system such as FAT, which is typically used on (older) desktop computers and servers, but also on thumb drives, memory cards and on internal memory of all kinds of embedded devices such as mobile phones and MP3 players. Interfacing with these devices often requires different hardware and accompanying software, but at some level the analyses converge and boil down to support for the FAT file system layout. Modularization can facilitate that each data format must be developed only once and then used in multiple scenarios on multiple devices.

Adding and Modifying Formats

All these independent implementations of standard data formats are rarely identical, prompting digital forensics investigators to regularly implement small changes or create derived versions of popular formats. Any implementation that is far removed from the specification of data formats will be difficult to use for regular adaptation.

For example, if a hand-written parser is used, making a small change such as changing the sign of all numbers in a data format can have significant impact on the entire implementation, such as having to change all the number variable

declarations and changing all calls to parsing methods related to numbers. Apart from being time consuming it is also error prone and difficult to verify.

Modifying and Reconfiguring Tools

The combination of diversity and similarity in the domain of digital forensics leads to additional complexity. An extremely rare combination of data formats in some areas may be very common in another. To analyze data efficiently, different investigations benefit from different combinations of algorithms and formats, each optimized for both a specific type and amount of data encountered.

An example is the analysis of the contents of a confiscated hard drive. In one investigation all files of certain types may be identified and recovered. In another however, time may be extremely limited and the investigators may be looking for a possibly hidden spreadsheet created using Microsoft Excel 2007. To accomplish this, they may want to look for all ZIP files containing XML files (since Excel 2007 files are basically a set of XML files compressed with ZIP). The more difficult it is to modify or reconfigure an application to perform this analysis, the less time the investigators will have to do other analyses.

Scalability

For the past thirty years, the cost of hard drive storage has shrunk exponentially as every fourteen months the price of a single gigabyte has halved [Kom09]. Coping with the amount of extra data would already be challenging in just this dimension, but there are more dimensions that show similar growth. The amount of households with broadband connections is steadily growing and in The Netherlands, there have been more active mobile phone subscriptions than citizens since 2006 [Cen09]. Additionally, the digital world is becoming more and more diverse, with desktops running Mac OSX and Linux operating systems slowly becoming more widespread and users choosing alternative browsers on any of these platforms are already common.

As a result, data analysis tools must scale to support these exponential increases in size as well as be able to identify and recover an increasing amount of different data formats.

Scaling To Terabytes

From a hardware perspective alone it is already challenging to have to analyse the largest hard drives available or network streams that do not fit on a single disk of the largest available size. The demands this places on the data analysis tools are even greater. Exponential growth in the encountered data means that analysis techniques and algorithms have to be extremely refined in order to be usable for

any length of time before they become too slow. When they do, it is typically a lot of work for developers to modify a data analysis tool to work with new techniques that have been optimized for the current generation of data sizes.

If the base functionality of these data analysis tools, such as reading and caching data as well as implementing identification and recovery algorithms is tangled with other concerns, especially related to identifying and recovering data formats, then every scalability enhancement will have to be applied to each data format implementation. This means that as data grows and more data formats come into use, not only will changes have to be made more frequently, they will also be more complicated every time. Eventually the data analysis tool will become unmaintainable.

Trading Precision for Speed

As mentioned in section 3.2, different types of investigations may be more efficient in a custom configuration using only a specific set of data formats and a single (type of) algorithm. However, there are also cases that this approach cannot be used to save time, for instance when there is not enough information about what to look for or how to look for it. When time is limited, a typical approach can be to simply reduce the precision of all parts of the system and end up with a best effort result given the time available.

If this requires a large amount of manual modifications across a large set of components, several problems arise. The first are typical for modifying software, such as making a lot of changes under time pressure being error prone and difficult to trace. Additionally however, a set of components developed by multiple developers across a large period of time will most likely consist of very different looking and functioning code, making it extremely difficult to modify all components in such a way that they all lose a comparable amount of precision and gain the same in performance. The result will be an unevenly optimized data analysis tool with difficult to predict performance characteristics.

3.3 A DSL for Digital Forensics

Specifying data formats is one of the main challenges identified in Section 2, so a data description language (DDL) [FMW10] forms our starting point. We have developed Derric, a DDL designed to address the problems related to data description in digital forensics. In the following subsection we will present the language using an example, the description of JPEG [ITU92]. The JPEG format is one of the most important data formats in digital forensics investigations, given that nearly all digital cameras and mobile phones produce files of this type and it is also the most prominent format for pictures on the world wide web.

```
format jpeg
extension jpeg jpg jfif

unit byte
size 1
sign false
type integer
strings ascii

sequence
  SOI
  APP0JFIF APP0JFXX?
  !(SOI APP0JFIF APP0JFXX EOI)*
  EOI

structures
SOI { marker: 0xFF, 0xD8; }
EOI { marker: 0xFF, 0xD9; }

Segment {
  marker: 0xFF;
  blockId;
  length: lengthOf(data) size 2;
  data: size length;
}

DHT = Segment { blockId: 0xC4; }
DQT = Segment { blockId: 0xDB; }

APP0JFIF = Segment {
  blockId: 0xE0;
  data: {
    identifier: "JFIF", 0;
    version: expected 1, 2;
    units: 0 | 1 | 2;
    xdensity: size 2;
    ydensity: size 2;
    xthumbnail: size 1;
    ythumbnail: size 1;
    rgb: size xthumbnail*ythumbnail*3;
  }
}

APP0JFXX = Segment {
  blockId: 0xDA;
  data: {
    identifier: "JFXX", 0;
    tnformat: 0x10 | 0x11 | 0x13;
    tndata: size length-
      (offset(tndata)-offset(length));
  }
}

SOS = Segment {
  identifier: 0xDA;
  comprData: jpegdata(ht=DHT.data,
                      qt=DQT.data);
}
```

Figure 3.1: Excerpt of the JPEG format in Derric.

An Example: JPEG

A Derric description is fully textual and consists of three parts: a header, a sequence and a set of structures. As an example, an excerpt of the JPEG image file format description is shown in figure 3.1.

Following is a discussion of how Derric addresses the domain-specific aspects of data description in digital forensics using the JPEG format description as an illustration.

Specification and Implementation Encoding

Derric allows literal values to be expressed in a large amount of different formats, tailored to different ways the data may be encountered in an investigation. In the case of reverse engineering, this will typically be in hexadecimal format. When documentation is used, other literals may be appropriate. The JPEG format description

in Figure 3.1 demonstrates several formats: line 21 shows hexadecimal and line 33 shows a string literal in combination with a regular decimal number. Additional formats are supported, including octal and binary.

In addition to the multiple formats for expressing values, modifiers exist to direct the interpretation of values. Modifiers exist to transform values based on byte ordering (little, big and middle endian), sign, numerical type (integer, float), string encoding (ASCII, UTF8, etc.) and size (with different units, such as bits and bytes). Default values for modifiers can be expressed at the top of a DERRIC description. An example of this is shown in lines 4–8. In this case, `"JFIF"` and `0` on line 33 will be interpreted as an ASCII string and a single byte, unsigned integer respectively.

Not requiring data format developers to transform data before use reduces the distance between actual data and data descriptions, thus improving usability and readability.

Expectations and Unknowns

Whether reverse engineering or working from documentation, some fields in a data format may have a lot of different values, but typically do not in practice. An example of this is the `version` field on line 34. Even though different versions of the JPEG format do exist, the 1.2 version is encountered nearly exclusively. Therefore, the value of the `version` field should formally be defined as any value. When attempting to reassemble a heavily fragmented JPEG file that has been cut off just before the `version` field however, it may improve performance dramatically to first try parts that start with the most common value for that field. The **`expected`** keyword in DERRIC allows the investigator to express this information as a hint to the recovery tool.

The opposite of having additional information about a field may also occur: not understanding the contents of a field completely and specifying whatever part is known or guessing. An example feature of DERRIC to facilitate this, is the **`terminatedBy`** keyword. It can be used for blocks of data that are not understood well enough to specify, by only specifying its terminator.

Allowing investigators to express additional or missing information about a data format as part of the specification enables an iterative style of development.

Modification and Variation

Decoupling the ordering into a separate sequence makes it easier to extend a description. Instead of specifying ordering at data structure level (e.g., as a linked list, which is common practice in many programming languages) a distinct sequence allows specifications such as on line 13, where the `!` operator used in `!(SOI APP0JFIF APP0JFXX EOI)*` automatically includes all data structures except

the ones specified. Adding a data structure automatically adds it to the sequence, which maps well to the process of reverse engineering where discovering previously unknown data structures is common. Additionally, if the **sequence** keyword is not specified, a sequence is inferred where any combination or ordering of specified data structures is accepted.

Data formats often have some fixed characteristics that are shared by most internal structures. In the case of JPEG, as shown by the `Segment` structure on lines 20-25 in Figure 3.1, this is a single byte `marker`, followed by a single byte `blockId`, following by a 16 bit `length` specifying the size of the data structure (in this case, apparently excluding `marker` and `blockId`) and finally the payload named `data`. Support in Derric for inheritance makes it easy to add another structure. As shown in the specification of the remaining structures on lines 27–58, inheritance allows overriding of fields (even with multiple fields, as shown on lines 32–41, where the `data` field is overridden by eight fields).

Decoupling the sequence from data structure specifications and inheritance make data descriptions shorter and help group related information, improving readability and expressiveness of the language.

3.4 Application: Carving

We have evaluated Derric in the domain of *file carving* [PM09], which is the process of recovering deleted, fragmented or otherwise lost files from storage devices. The complete description of Figure 3.1 has been input to a code generator to obtain a JPEG validator. Such a validator can be used by dedicated carving algorithms [Gar07] to recover evidence from disk images. The complete system including file format descriptions in Derric, code generator and runtime library is named Excavator.

Concerns in the Carving Domain

Analysis of the carving domain uncovers three concerns that are variable across typical carver implementations: (1) Format, (2) Matching and (3) Reassembly. A schematic overview of this variability is shown in Figure 3.2. The first type of variability entails that for each type of file that must be recovered, the file format must be defined. Carvers must know the structure of, for instance, JPEG in order to recognize that a certain sequence of bytes might be part of a valid JPEG file. Additionally, some file formats exist in different versions and variants. For instance, the Portable Network Graphics (PNG) format has three official versions [W3C03]. Finally, manufacturers of digital devices such as mobile phones or digital cameras may implement a file format standard in idiosyncratic ways, which could be valu-

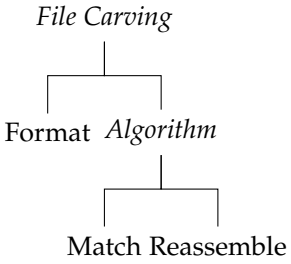

Figure 3.2: Variability in the file carving domain.

able for recovery. We consider all kinds of variation to be covered by the "Format" concern.

The second dimension captures (1) the ways in which files are matched in the input image, and (2) the method of reassembly if fragmentation is detected on the basis of file format structure. In Figure 3.2 these variation points are indicated as "Match" and "Reassemble" respectively, below the abstract "Algorithm" concern.

There are at least three matching algorithms that are used in carvers. The most basic matching algorithm is header/footer matching that returns blocks between signatures of file headers and footers. Next, file structure-based matching uses complete structural knowledge of a file format in order to deal with, for instance, corrupted files. Finally, characteristics-based matching takes (statistical) characteristics about a file's contents into account, for instance high entropy in compressed files.

Finally, the third concern consists of algorithms for reassembling fragmented files. For instance, bifragment gap carving [Gar07] assumes that files consist of only two fragments and that they are located on the data storage device in the correct order. The algorithm tries all possible gaps between the matched beginning and end of the file. Map/generate [Coh07] is more elaborate in that it supports reassembling files that are arbitrarily fragmented. It exercises any combination of sectors and then prunes the search space if mismatches are found.

Currently, file carvers implement a limited combination of file formats and/or matching and/or reassembly algorithms. Off-the-shelf carvers typically do not support explicit variation points to efficiently make trade-offs between precision and performance. The implementation of data format, matching and reassembly is completely tangled. As a consequence, modification or reconfiguration of carvers is time consuming and error prone.

Additionally, the top-level dimensions of Figure 3.2, "Format" and "Algorithm", correspond to two different roles in the practice of using carvers in forensic inves-

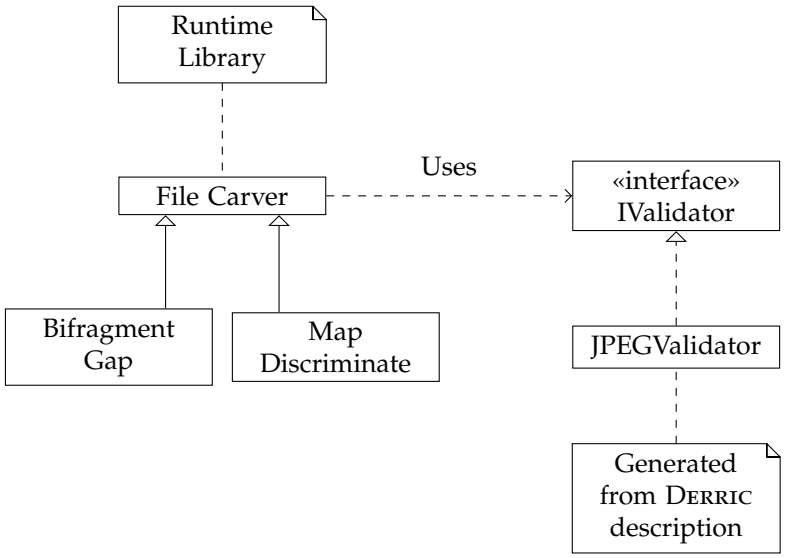

Figure 3.3: Overview of the Excavator architecture.

tigations. On the one hand there are the digital forensics investigators that have intricate knowledge of many file formats. On the other hand, there are the software engineers that know how to implement, evolve, and optimize carving tools. With the current tools, no division of labour is possible: domain-specific knowledge about file formats has to be communicated to software engineers in order for them to make the necessary changes to the system.

Each concern of Figure 3.2 corresponds to a variation point in the implementation. In Excavator, each variation point corresponds to a logical component. These components are:

1. The declarative surface syntax of Derric for describing the structure of file formats (*Format*).

2. A code generator that takes file format descriptions and generates matching code (*Matching*)

3. A runtime library implementing reassembly algorithms as well as defining the base types and interfaces for the generated matching code. (*Reassemble*)

Both the file format metamodel and the code generator are implemented in Rascal [KSV09a]. File format descriptions are input to the code generator. The generator produces Java classes implementing the "Matching" concern. These classes are

Tool	Version	Command line
ReviveIt	20070804	`-e -F -t OUTDIR -c ../etc/file_types.conf INPUT`
Scalpel	1.6	`-b -c scalpel.conf -o OUTDIR INPUT`
PhotoRec	6.11	`/d OUTDIR INPUT`

Table 3.1: Carvers participating in the evaluation.

used by the Java runtime library which contains algorithms for fragment reassembly. Currently, the runtime library contains two algorithms, a brute force algorithm and bifragment gap carving discussed in Section 3.4.

The final component is the code generator. It takes a description such as that of Figure 3.1, and produces a Java class implementing the matching code that is used by the runtime library. This code generator uses a model-to-text approach [CH06]. It is implemented using Rascal's string templates, which are ordinary strings, interpolated with arbitrary expressions and control flow statements.

An overview of the Excavator architecture is shown in Figure 3.3. The abstract Carver class captures the reassembly concern; implementations exist in two variations as indicated by the concrete subclasses. A carver uses implementations of the IValidator interface (matching concern). Implementations of this interface are generated from Derric file format descriptions.

Evaluation

In order to evaluate the resulting JPEG carver, we have compared its performance to that of three popular carvers. First, we assert that Excavator is in the same league with respect to the number of recovered files and runtime performance. For this, the carvers are run on five established benchmarks for carvers. Secondly, we argue that the flexibility induced by the domain-specific language approach of Excavator is unmatched by the other carvers.

The file carvers that we compare Excavator to were chosen based on two criteria. First, they are actively used in digital forensics investigations, both in government and industry. This ensures our comparison is relevant. Second, we required the tools to be open source in order to make a source-based assessment of the effort of customizing a carver. These criteria have lead to the selection of Scalpel [RR05], PhotoRec [Gre09] and ReviveIt [Met]. The precise versions and command line options that were used in the evaluation are shown in Table 3.1. Below, we briefly describe each carver.

Scalpel A high performance, file system-independent and cross-platform file carver

#	Short name	Name	Size (MB)	#JPEGs
1	JPEG 1	JPEG Search Test #1	10	7
2	Basic 1	Basic Data Carving Test #1	62	3
3	Basic 2	Basic Data Carving Test #2	123	3
4	DFRWS'06	Forensic Challenge 2006	48	14
5	DFRWS'07	Forensic Challenge 2007	331	18

Table 3.2: File carving tests participating in the evaluation.

written in C. It employs a header/footer based algorithm to recognize files; the structure of headers and footers is described using regular expressions. It tends to generate a relatively large amount of false positives but is extremely fast.

PhotoRec Originally designed to recover digital photographs from memory cards but has since been extended to support a plethora of file formats. This carver is completely implemented in plain C and all logic, file format, matching and reassembly, is hard-wired.

ReviveIt The most advanced of the three carvers. It employs Garfinkel's bifragment gap algorithm [Gar07] and is configured using an external specification of file formats. This specification is then interpreted at runtime.

Forensic Benchmarks

The set of benchmarks used in the evaluation of Excavator consists of five files containing either a byte-for-byte copy of a data storage device or a synthetic data structure with similar properties. The files were selected since they all contain recoverable JPEG files and are widely recognized as benchmarks for carvers.

The size of each benchmark, together with the number of JPEG files contained in it, is shown in Table 3.2. JPEG 1, Basic 1 and Basic 2 originate from the *Digital Forensics Tool Testing Images* [Car] collection, a project set up to share benchmarks that are useful for testing digital forensics tools. They are regularly used to evaluate new algorithms and tools in digital forensics research. DFRWS 2006 and DFRWS 2007 are taken from the Digital Forensics Research Workshop's (DFRWS) Forensic Challenge in 2006 and 2007, when the challenge focused on file carving. Together, the benchmarks exercise file carvers in nearly all relevant areas, such as recovering deleted files, reassembling fragmented files, ignoring placed false positives and dealing with file system-specific issues. Below we briefly describe each benchmark.

	ReviveIt		Scalpel		PhotoRec		Excavator	
JPEG 1	4	(57.1%)	6	(85.7%)	4	(57.1%)	6	(85.7%)
Basic 1	3	(100%)	1	(33.3%)	3	(100%)	3	(100%)
Basic 2	3	(100%)	1	(33.3%)	1	(33.3%)	3	(100%)
DFRWS'06	10	(71.4%)	6	(42.9%)	8	(57.1%)	8	(57.1%)
DFRWS'07	1	(5.6%)	0	(0%)	1	(5.6%)	1	(5.6%)
Total	21	(46.7%)	14	(31.1%)	17	(37.8%)	21	(46.7%)

Table 3.3: Number of true positives and recall per carver, per benchmark.

JPEG Search Test #1 An NTFS file system containing JPEG files in various disguises. Additionally, some traces of JPEG headers and footers have been placed in strategic locations, to confuse carvers.

Basic Data Carving Test #1 A byte-for-byte copy of a 64MB FAT 32 formatted thumb drive including deleted files and some corrupted data structures, including a JPEG header.

Basic Data Carving Test #2 A byte-for-byte copy of a 128MB EXT2 formatted thumb drive including deleted and fragmented files.

DFRWS Forensic Challenge 2006 A 50MB file generated using random data and seeded with, amongst others, fragmented JPEG files which may be interleaved with other JPEG files or hand-crafted headers to confuse carvers.

DFRWS Forensic Challenge 2007 Similar to the 2006 DFRWS benchmark, only larger (331MB) and heavily fragmented.

Evaluation Details

To compare the existing carvers to Excavator, we have run all four carvers on all five benchmarks. To ensure the best results, if a tool has multiple modes of operation we have run each benchmark in each mode and recorded the best result—see Table 3.1 for details on how each carver was run.

File Carving Performance

Table 3.3 lists the results of our evaluation. The table shows the number of correctly recovered files for each carver, including the recall between parentheses.

Of all carvers, Scalpel recovers the smallest amount of files. The reason is that its simple header/footer matching algorithm prevents it from recovering any fragmented files. PhotoRec performs better, but does not find files that are prefixed

with random data (JPEG 1) and has trouble dealing with fragmentation in the EXT2 benchmark (Basic 2). ReviveIt also misses the files that are prefixed with random data (JPEG 1), but does succeed in reassembling more fragmented files than any other tested carver (DFRWS 2006) through its combination of file structure matching, characteristics-based matching and bifragment gap reassembly. Finally, Excavator recovers several fragmented files as well but misses a few more than ReviveIt because it does not implement characteristics-based matching. However, it does recover the random data prefixed files (JPEG 1).

Table 3.3 shows the number of files that (1) are completely recovered and (2) are actually present in the test image. The first condition is checked by feeding the recovered file to an image viewer. The second condition is verified using the MD5 checksums provided with each benchmark. Any file that is recovered, but is not viewable or does not match an MD5 checksum, is a false positive.

We chose not to include the number of false positives (and hence, the precision) in the results for two reasons. First, when a file matches none of the MD5 checksums, it is not automatically useless in forensic investigations. For instance, it may be a partial file containing crucial evidence. Thus, a false positive is not necessarily a bad thing. Second, the degree as to which a false positive is useless, is hard to quantify. During our experiments, we have observed that some files were partially recovered by multiple tools, but that some tools recovered a larger part than others. Which part of a file is important depends on the case at hand. We have therefore chosen to only measure the number of true positives and recall[1].

Nevertheless, a large number of false positives is not desirable, since they have to be manually inspected. In all of our tests, Excavator had no more false positives than the best performing tool of all the tools in the evaluation.

From the results it can be concluded that Excavator, on average, finds as many files as the other carvers. In fact, the only benchmark where Excavator performs worse than any other carver is DFRWS 2006: ReviveIt recovers two more files because it employs characteristics-based matching. We expect that adding support for characteristics-based matching to Excavator will make it as good as ReviveIt on DFRWS 2006 as well.

Runtime Performance

The runtime performance results are shown in Table 3.4. On the whole, ReviveIt performs worst on all benchmarks. This can be explained by its use of characteristics-based matching, which requires it to process much more data than the other tools. The other tools typically finish within a couple of seconds with a few exceptions.

The high running times of ReviveIt and Scalpel on DFRWS 2007 can be explained from the fact that the image is much larger than the others, and both tools recover

[1] This is in accordance with the rules used in the DFRWS Forensic Challenges.

	JPEG 1	Basic 1	Basic 2	DFRWS'06	DFRWS'07
ReviveIt	11.8s	14.6s	17.8s	37.0s	7m58s
Scalpel	0.4s	1.9s	3.6s	2.8s	21.7s
Photorec	0.2s	0.5s	0.6s	0.2s	3.8s
Excavator	0.2s	0.4s	0.8s	18.6s	3.1s

Table 3.4: Runtime performance per carver, per benchmark (wall clock time).

many partial files. On the DFRWS 2006 benchmark, ReviveIt and Excavator use bifragment gap carving to recover some fragmented files that both Scalpel and PhotoRec miss; this explains the additional time required.

Based on the numbers in Table 3.4, we conclude that the runtime performance of Excavator is similar to the performance of the fastest carvers in these benchmarks. Important to note however is that in real-life digital forensics investigations, the data sets will typically be much larger, since hard drives of several terabytes in size are becoming common. Unfortunately, no publicly available benchmarks exist of this size. As a result, we have not been able to determine how Excavator scales compared to the other carvers.

Flexibility

Efficiently implementing new file formats or modifying existing ones is an important requirement in digital forensics investigations. The carvers in our evaluation all support this requirement with varying degrees of flexibility. Below we provide a qualitative assessment of the domain-specific language approach of Excavator in comparison to the other carvers.

PhotoRec requires a file format definition to be directly implemented in code, along with a matching algorithm. This tangling of concerns makes it practically impossible for a non-programmer to make changes. Furthermore, to leverage advances in matching algorithms, existing file format implementations must be adapted.

Apart from PhotoRec, all other carvers have separate file format definitions that can be modified without altering the application code. The definition that Scalpel uses, however, is very basic: header and footer matching along with some basic options (such as case sensitivity). This means that the built-in header/footer matching is hard to replace with a more advanced matching algorithm. Furthermore, reassembly algorithms typically require the matching to be much more precise in order to do scalable reassembly.

Component	Implementation	Size (SLOC)
Grammar	SDF	52
JPEG description	Derric	92
png description	Derric	58
Structure-based matching (code generator)	Rascal	510
Bifragment gap (runtime)	Java	72
Brute force (runtime)	Java	44
Utilities (runtime)	Java	256
	Total:	1084

Table 3.5: Sizes of the Excavator components.

The remaining two, ReviveIt and Excavator, support full file format descriptions. The definition that ReviveIt uses however is tied to concepts of the matching algorithms it implements. For instance, its definitions mention characteristics-based matching, which in our view belongs to the matching concern and not to the definition of a file format. As a result, these file format definitions are hard to reuse for alternative matching algorithms and even harder for different types of data analysis. Excavator's file format definitions are strictly declarative; both matching algorithm and file formats can be varied independently.

Furthermore, Excavator separates matching and reassembly algorithms, allowing variation between these dimensions in a similar manner. None of the other carvers expose explicit variation points to independently vary matching and reassembly[2]. Excavator can be run using both bifragment gap and brute force reassembly algorithms without having to adapt file format descriptions.

The results of Tables 3.3 and 3.4 show that the separation of concerns achieved in Excavator did not incur a penalty in either carving performance or runtime performance. Moreover, this flexibility did not come at the price of more code either. Table 3.5 shows the size statistics of Excavator. The entire system, currently encompassing the language grammar, JPEG and PNG descriptions, code generator and a runtime library containing two reassembly algorithms, consists of just above a thousand non-commented lines of source code.

3.5 Discussion

Although techniques such as separation of concerns and declarative specification are commonly regarded as improving quality whenever they are used, it is diffi-

[2]Scalpel does not support reassembly at all.

cult to assess whether any given solution applies these principles completely, correctly and whether an even better solution could exist. Nonetheless, given the very small amount of code required to develop Excavator and the results achieved, we believe the general effectiveness of the approach is clear. However, some issues surrounding suitability and applicability exist and are addressed in the following subsections.

Scalability

One of the challenges discussed in Section 2 is improving scalability. To measure this, benchmarks or scenarios must be used that push a data analysis tool to the limit in terms of data size it can handle. However, the largest publicly available benchmark is the DFRWS Forensic Challenge 2007 image, which is included in our test set. At 331MB, this does not come near a size that requires an analysis tool to take special measures in the area of scalability. This challenge therefore has not been addressed.

Universal Data Description

Derric can be used to describe any data format, but in the current evaluation has only been used to describe JPEG. The language's usability however depends on its ability to describe a large range of data formats. In order to develop our language, we have described a large set of data formats, including other image formats such as PNG and GIF, along with several document formats such as Microsoft Office Word and Excel and container formats such as ZIP and RAR. To test our language and code generator, these descriptions were successfully tested on sets of files of those types. The benchmarks are all focused on JPEG, so our measurements use these results.

The application we have developed recovers data but does not process it further. Additional capabilities that may be related to the data description language, such as processing embedded files in a container format or detecting encryption are therefore not evaluated. However, recognizing a file's type without processing it further is useful outside of data recovery, for example in network filtering and content detection (in network proxies and web browsers).

Usability

The eventual users will be the final judge of Derric's usability. Even though we do not have numbers on user satisfaction, we believe that there are several reasons that Derric can be considered an improvement over other approaches. First, when data format processing code is developed by a software engineer, the digital forensics investigator would need to transfer knowledge of a data format to the engineer.

Derric provides a tailored notation that can be directly used by digital forensics investigators.

Second, if the investigator develops the data format processor directly, then Derric still only requires the same information that would otherwise need to be expressed in any programming language, but stripped of all implementation details such as memory management. Therefore we believe Derric can be considered a step forward from direct implementation since it requires nothing more, but does remove a lot of work for the investigator.

3.6 Related Work

There is extensive work in the area of model-driven engineering (MDE) [Sch06], DSLs [DKV00] [Spi01] [KT08] and DDLs [FMW10].

The work in [Sta06] investigates the factors that influence industrial adoption of MDE. One of the conclusions is that generic, well-established modeling languages are favoured over more advanced modeling technologies, such as dedicated DSLs. As such, this identifies an open research question regarding our work.

A case-study of MDE in an industrial context is described in [BLW05]. There, the use of MDE has been found to lead to significant productivity and quality improvements. In one division that was investigated 65%–85% of the code could be generated from high-level models. Moreover, the model driven perspective also lead to improvements in some phases of the software process: it turned out that the time to correctly fix a defect was regularly reduced by a factor between 30 and 70. This tremendous gain is attributed to the fact that many defects could be fixed and tested at the model level. This can be seen as additional supporting evidence for the observation that MDE may significantly improve changeability of software.

In [MHS05] a survey of the techniques and tools related to the different stages of DSL development is presented. One important conclusion is that these nearly all focus on the implementation phase and ignore earlier phases such as decision, analysis and design.

A study of the success factors of DSLs is described in [HPD09]. There, learnability, usability and expressiveness of the DSL, reusability of the code and development costs and reliability of the resulting software are identified as the most important factors contributing to the success of using a DSL.

Most data description languages are either tied to a specific type of application, such as PacketTypes [MC00] and Zebu [BRLM07] to network protocols, or technology, such as XML Schema to XML.

Some general data description languages that allow specification of binary formats do exist, such as PADS [MFW+07] and DataScript [Bac02]. Of these, PADS supports extensive error handling. Derric distinguishes itself by having a syntax

that maps onto common activities in the field of digital forensics such as reverse engineering.

The technology behind file carving is strongly related to parsing [GJ08]. However, traditional grammar formalisms, such as ANTLR [Par07] and SDF2 [Vis97], are specifically targeted at describing textual computer languages. They are generally unsuitable to build parsers for binary file formats, since these often require complex data dependencies between elements of a file. Data-dependent grammars extend traditional parsing technology to allow the definition of such dependencies [JMW10] and may be usable in some applications of DERRIC.

3.7 Conclusion

Data storage size and network bandwidth is growing continuously and popularity of digital hand-held devices is increasing. Additionally, the software market is diversifying and growing steadily. This brings serious challenges to digital forensics investigators who must cope with large quantities of data and an evolving set of data formats to consider.

We present a practical interpretation in the area of software engineering of these digital forensics challenges, identifying which activities are directly affected by them, so they can be addressed systematically.

Next, we present the domain-specific language DERRIC, designed to fit into the workflow of a digital forensics investigator. It allows declarative specification of data formats, thus separating the task of data description from data analysis tool development, enabling increased data abstraction and modularization.

To evaluate DERRIC we have developed EXCAVATOR, a data analysis tool in the area of file carving making full use of DERRIC to describe data formats. EXCAVATOR is compared to popular existing file carvers used in practice on a test set consisting of standard carving benchmarks and challenges used in digital forensics research. Our comparison shows that EXCAVATOR is in the same league as the existing file carvers in terms of carving results and runtime performance, while requiring minimal effort to develop and allowing reuse of its data format specifications.

Directions for Future Work

In order to better validate our efforts to address the challenges in the areas of data abstraction and scalability, we intend to develop a test set that is large enough to allow evaluation of scalability and contain a large amount of files in different data formats that are representative of the domain, including image, movie, document and container formats.

Second, to evaluate whether our language is usable in multiple areas of digital forensics, we intend to develop different data analysis tools based on Derric, for instance to analyze network streams and memory layouts.

Finally, a user evaluation of Derric must be performed among actual users of the language, once a system using it is actually deployed and used in real-world digital forensics investigations.

CHAPTER 4

Domain-Specific Optimization in Digital Forensics

This chapter was previously published as a paper with the same title in *5th International Conference on Model Transformation (ICMT'12)* [BS12]. Joint work with Tijs van der Storm.

Abstract

File carvers are forensic software tools used to recover data from storage devices in order to find evidence. Every legal case requires different trade-offs between precision and runtime performance. The resulting required changes to the software tools are performed manually and under the strictest deadlines.

In this chapter we present a model-driven approach to file carver development that enables these trade-offs to be automated. By transforming high-level file format specifications into approximations that are more permissive, forensic investigators can trade precision for performance, without having to change source.

Our study shows that performance gains up to a factor of three can be achieved, at the expense of up to 8% in precision and 5% in recall.

4.1 Introduction

Digital forensics is a branch of forensic science that attempts to answer legal questions based on the analysis of information recovered from digital devices. These digital devices are typically computers or mobile phones confiscated from a suspect, found near a crime scene or otherwise expected to have information stored that is relevant to an investigation. In the context of this chapter we are interested in *file carvers*: tools that recover data from storage devices without the help of (file system) storage metadata [PM09].

The current growth in size of storage devices requires that file carvers scale to analyze data in the terabyte range. Moreover, forensic investigations are often performed under very strict deadlines, making the runtime performance of such tools critical. Additionally, the large diversity in (variants of) file formats encountered on devices requires these tools to be easy to modify and extend.

Because each case may require different trade-offs with respect to precision and runtime performance, file carvers often need to be modified on a case-by-case basis. Currently, this kind of just-in-time "carver hacking" is performed by hand, which is error prone and time consuming; it is also inherently incompatible with very strict deadlines.

In previous work we have developed a model-driven approach to digital forensics tool construction (see Chapter 3). In this work the file formats of interest, e.g., JPEG, GIF etc., are declaratively modeled using a domain-specific language (DSL) called Derric. These descriptions are then input to a code generator that produces highly efficient and accurate format validators that form an essential part of our file carver Excavator.

Excavator competes with file carvers widely used in practice, and is much easier to maintain due to the high-level Derric language. Nevertheless, the generated components encode a particular trade-off between precision and runtime performance. In this work we apply model transformations on Derric descriptions in order to make this trade-off configurable. We present three model transformations that successively obtain format validators that are more permissive (i.e., produce more false positives) but exhibit better runtime performance. As a result forensic investigators can choose between precision and runtime performance without having to change any code.

We have evaluated Excavator using the different format validators at each permissiveness configuration for the file formats JPEG, GIF and PNG on a representative test image of 1TB. Our results show that performance gains up to a factor of three can be achieved, at the expense of up to 8% in precision and 5% in recall.

This chapter makes the following contributions:

- We present three model transformations to automatically derive format val-

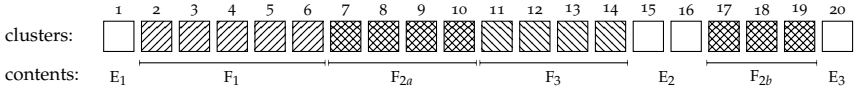

Figure 4.1: An example set of contiguous clusters on a storage device.

idators that trade precision for better runtime performance.

- We evaluate our approach on a representative test image in the terabyte range showing that substantial performance gains can be achieved.

Organization of this chapter

The rest of this chapter is organized as follows. Section 4.2 discusses file carving and analyzes the development, performance and scalability challenges in the engineering of digital forensics software. We introduce our model-driven approach to building file carvers and discuss how it addresses the challenges. This includes an overview of DERRIC, our domain-specific language (DSL) for file format description. Section 4.3 defines three model transformations on DERRIC descriptions. Section 4.4 evaluates the effect of the model transformations on the runtime performance and precision of the generated carvers. In Section 4.5 we discuss our results. Related work is discussed in Section 4.6. We summarize our research and results in Section 4.7.

4.2 Background

File Carving

When recovering data from a storage device, all available metadata such as file system records and application logs are used to identify locations where data is stored. After this initial step, there is usually a significant amount of *unallocated space* left on the storage device. This space may contain only zeros (or some other factory default value), but may also contain deleted files, operating system caches or data that has been hidden on purpose. To recover this data, a content-based technique called *file carving* can be used.

A typical modern file carver consists of a set of format validators used by one or more file reconstruction algorithms. In its most basic form the format validators consist of checking for format-specific constants at the start and end of a stream (called *header/footer matching*) and the file reconstruction algorithm simply moves through the input stream in a single pass, invoking all format validators at each

offset to determine whether a file is located there. On each hit, the identified file is saved for further analysis.

Apart from generating a large amount of false positives, this approach has another drawback: it is unable to recover files that are split into multiple parts and stored in non-contiguous locations. This so-called file fragmentation is common, usually as a result of performance optimization by the operating system and implementation details of the file system.

To recover fragmented files but avoid a combinatorial explosion, file carvers implement file reconstruction algorithms, such as bifragment gap carving [Gar07]. However, to improve precision and reduce the amount of required iterations to reconstruct a single file, they also use more advanced format validators that validate (part of) the format's structure and content.

Common optimizations include running multiple format validators on the same block of data concurrently and applying data classification techniques to reduce the search space (e.g., removing blocks of zeros). These techniques are not discussed further in this chapter.

File Carving Example An example set of contiguous clusters commonly found on storage devices is shown in Figure 4.1. Clusters 1, 15, 16 and 20 contain only zeros. The remaining clusters contain three files: F_1 (clusters 2–6), F_2 (fragmented, clusters 7–10 and 17–19) and F_3 (clusters 11–14).

A traditional file carver that performs a single pass over the data checking for headers and footers only will probably recover F_1, since it will find a header in cluster 2 and a correct following footer in cluster 6. Fragmented file F_2 is problematic, as the first footer following the header in cluster 7 is F_3's footer in cluster 14. As a result, both F_2 and F_3 are not recovered.

A more sophisticated format validator may detect a problem around cluster 11 or 12 and report this to the file carver. The file carver can then decide to look for suitable footers within a certain range, possibly finding both F_3's footer in cluster 14 as well as F_2's footer in cluster 19. Some shuffling of the clusters between the original error location in cluster 11 and the potential footers may lead the file carver to consider clusters 7–10 and 17–19, which the format validator will accept. From the remaining clusters, F_3 will then be easy to recover as well.

Software Engineering Challenges

From a software engineering perspective, the challenges in file carver construction can be classified into three areas, described in the following subsections.

Modifiability

Digital forensics tools must be continually adapted to new versions and variants of storage formats encountered during investigations. For instance, even when using a standardized format such as the JPEG image file format, different vendors of, for instance, digital cameras may store the actual files in different ways, often deviating from the standard. When forensic investigators encounter traces on some device that they want to recover or analyze, they often need to adapt their tools to these new, modified or different storage formats in order to maximize recoverable evidence.

Runtime Performance

Strict time constraints means that analyses must be completed as quickly as possible, even when the amount of data to analyse grows very fast. Brute force algorithms are intractable when it comes to reconstructing a file by finding its parts in a set of millions of fragments. Hence, the challenge is to use as much domain-specific knowledge as possible for optimization. This includes knowledge about hardware, operating systems, file system implementation, file formats and typical fragmentation patterns [Gar07].

Scalability

Digital forensics tools must be scalable to deal with relatively large data sizes. Common hard drive sizes in desktop computers are already in the terabyte range. Support for these data sizes imposes additional constraints on the design and implementation of tools. Recovering evidence from a set of data of which 1% barely fits into working memory requires custom approaches. Most analyses must use a streaming architecture to collect information while reading through the data from beginning to end in a single pass.

Model-driven Digital Forensics

To address the challenges described in the previous section, we have developed a model-driven approach to file carver construction, called Excavator. The architecture of Excavator consists of three parts and is shown in Figure 4.2.

The first part is a domain-specific language called Derric that allows file formats to be specified in a declarative way. A simplified example of a Derric specification of the PNG image file format is shown in Figure 5.1, which will be discussed in more detail below. A Derric file format description captures the information to be used by a file carver to recognize (fragments of) files in a data stream. Derric file format descriptions are tailored to digital forensics applications; they may leave

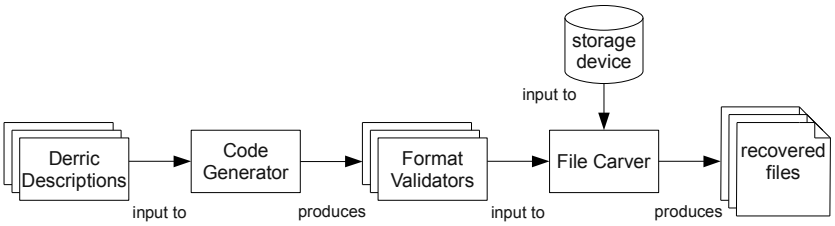

Figure 4.2: Overview of the Excavator architecture.

out details of a file format that would be relevant for implementing a file viewer, for instance, but are not important for file carving.

The Derric file format descriptions are input to the second component, a code generator to obtain format validators. A format validator is used to check that a certain sequence of bytes indeed can be recognized as part of a file format. The code generator performs domain-specific optimizations to make the resulting code as efficient as possible, such as skipping over blocks of data that will not be interpreted and only generating variables for values read from the input data that will actually be referenced. Both the Derric DSL[1] and the Excavator code generator have been developed using Rascal[2], a DSL for source code analysis and transformation [KSV09a]. The code generator produces Java source code.

The third part is the file carver itself, which employs dedicated algorithms and heuristics for locating candidate files in the data stream. This component uses the generated format validators to verify if a candidate file is an instance of a file format. This component can be considered the runtime system of Excavator. The runtime system is implemented in Java using the latest IO libraries for maximum throughput.

Excavator can be configured to run with or without file reconstruction capabilities. The algorithm it implements is bifragment gap carving with a configurable maximum gap size, with a default value of 2MB. It supports a variable cluster size with a default value of 4096 bytes. It does not support parallelism or filtering through data classification.

Our model-driven approach to digital forensics tool construction addresses the

[1] http://www.derric-lang.org/
[2] http://www.rascal-mpl.org/

aforementioned challenges in the following way:

- **Modifiability** Using high-level file format descriptions separates the "what" from the "how": if a new variant or version of a file format has to be accomodated, only the file format description has to be changed; the code generator and runtime system remain unchanged.
- **Runtime performance** The code generator can apply sophisticated optimizations to obtain fast code. Because this concern is now isolated in the code generator, it does not affect the description of file formats. Traditionally, optimizations in digital forensics tools are tangled with the matching logic of file format structure.
- **Scalability** The runtime system effectively captures the way data is processed, independently from the generated validators. This means that a file carver can be made to run in streaming fashion by changing the runtime system. Additionally, state-of-the-art file carving algorithms (e.g., [Coh07]) can be plugged into the system without affecting the other components.

Still, there is room for improvement. Digital forensics tools are often adapted to a certain situation in order to trade quality and completeness of the results for increased performance. On the one hand, if a recovery tool produces many false positives, this may be problematic, because they all have to be inspected manually. On the other hand, this may be preferable to not having any results at all before the deadline. In order to make this trade-off configurable we can apply model transformations to Derric file format descriptions to obtain a faster file carver at the cost of some precision. These transformations are described in Section 4.3.

Example: PNG Image File Format

As an illustration of Derric, we present a description of a simplified version of the PNG image file format in Figure 5.1. It omits the details of optional data structures but is complete enough to be transformed into a validator that properly recognizes PNG files.

At the beginning of the format description, the name of the format is specified (line 1) along with a set of storage-related defaults, such as string encoding (line 2) and default numerical type (lines 3–6), in this case single-byte unsigned integers.

Next is the definition of the format's `sequence` (lines 8–11), which defines the ordering of data structures in a valid file. In this example only a single operator appears (asterisk), which specifies that the structure must appear zero or more times. Additional constructs exist such as selection (parentheses), subsequencing (square brackets), optionality (question mark) and exclusion (exclamation mark).

```
1 format PNG
2   strings ascii
3   sign false
4   unit byte
5   size 1
6   type integer
7
8 sequence
9   Signature IHDR
10  Chunk* IDAT IDAT* Chunk*
11  IEND
12
13 structures
14 Signature {
15   marker: 137,80,78,71,13,10,26,10;
16 }
17
18 Chunk {
19   length: lengthOf(chunkdata) size 4;
20   chunktype: !"IDAT" size 4;
21   chunkdata: size length;
22   crc: checksum(
23         algorithm="crc32-ieee",
24         start="lsb",store="msbfirst",
25         fields=chunktype+chunkdata)
26        size 4;
27 }
28 IHDR = Chunk {
29   chunktype: "IHDR";
30   chunkdata: {
31     width: !0 size 4;
32     height: !0 size 4;
33     bitdepth: 1|2|4|8|16;
34     colourtype: 0|2|3|4|6;
35     compression: 0;
36     filter: 0;
37     interlace: 0|1;
38   }
39 }
40
41 IDAT = Chunk {
42   chunktype: "IDAT";
43   chunkdata: compressed(
44                 algorithm="deflate",
45                 layout="zlib",
46                 fields=chunkdata)
47              size length;
48 }
49
50 IEND {
51   length: 0 size 4;
52   chunktype: "IEND";
53   crc: 0xAE, 0x42, 0x60, 0x82;
54 }
```

Figure 4.3: Structure of the simplified PNG image file format.

The final part is the **structures** block (lines 13–54), defining the structures mentioned in the **sequence**. Each structure has a name and a list of field descriptions between curly braces. For example, the `Chunk` structure on lines 18–27 has four fields: `length` (line 19), `chunktype` (line 20), `chunkdata` (line 21) and `crc` (lines 22–26).

The `Chunk` structure's fields demonstrate some of Derric's specification constructs. The `length` field has the length of the `chunkdata` field as value, and its type is a 32-bit unsigned integer. The `chunktype` field is four bytes in size and may contain any value except the ASCII string "IDAT". The `chunkdata` field does not specify its value but constrains that its size must correspond to the value of the `length` field. Circular references like this are common in format descriptions and are useful in situations where only part of a data structure has been recovered; each value can be used to validate the other.

Finally, the `crc` field has a fixed size of four bytes and defines a value that must be calculated using the "crc32-ieee" algorithm (line 23) using the values of the `chunktype` and `chunkdata` fields (line 25).

Additionally, Derric supports structure inheritance. This is shown on line 28

where the `IHDR` structure inherits the fields of the `Chunk` structure and then overrides the `chunktype` and `chunkdata` fields (lines 29–38). Its `length` and `crc` fields remain the same as in `Chunk`.

4.3 Transforming Derric Models

In order to make the trade-off between precision and runtime performance configurable we have implemented three model-transformations on Derric descriptions, based on an analysis of validation techniques in file carving (see Chapter 2). Each transformation removes constraints so that more permissive specifications are obtained. The transformations consist of replacing computationally expensive operations with cheaper versions that resemble the original technique, or skip over data entirely instead of processing it. They can be applied successively so that in the end four format validators can be derived from a Derric specification. The transformations are source-to-source transformations; as a result, the generic code generator of Excavator can be reused to obtain a working format validator from each transformed description.

Using the transformations, we can distinguish four configurations of format validator precision:

- **Base**: base validator (the most precise validator, based on the complete file format description).
- **NoCA**: removal of all content analysis (e.g., removal of CRC checks, data decompression, etc.).
- **NoDD**: removal of all data dependencies (e.g., a field's value becomes undefined if it used to be equal to the contents of some other field's value).
- **Header**: removal of all matching except header and footer patterns.

Although each transformation could be applied independently, for the purpose of this chapter we only consider the consecutive application of each transformation. The effect of other combinations of transformations is left as future work. The transformations are described in more detail below.

Remove Content Analysis

The most computationally expensive technique is content analysis, which is the interpretation and validation of a file's content, as opposed to matching structural metadata. For instance on lines 22–25 of Figure 5.1 a CRC_{32} over each `Chunk` of PNG data is defined using the **`checksum`** keyword. Additionally, lines 43–46 describe the compression scheme used by the `IDAT` structure using the **`compressed`** keyword.

Removing these expensive analyses will reduce running time significantly at the cost of missing some fragmented files due to lower precision.

Removing content analysis consists of one of two rewrites, based on the field the content analysis is defined on:

- If the field has an externally defined size, i.e., if it has a fixed value (such as the CRC32's four bytes) or references an outside value (such as the `IDAT`'s reference to its `length` field), the field's value specification is removed. As a result, the data will be skipped over instead of processed.
- When the end of a field is specified by an end marker as part of the content analysis itself, the end marker is lifted out of the content analysis specification to be used to specify the end of the field.

More precisely, the transformation is defined by the following two rules:

$$f\texttt{: } CA(\overline{x})\ \texttt{size}\ n\texttt{;} \Rightarrow f\texttt{: size } n\texttt{;}$$
$$f\texttt{: } CA(\overline{x}\texttt{, terminator=}c\texttt{);} \Rightarrow f\texttt{: terminatedBy } c\texttt{;}$$

The first rule replaces a fixed-length field f which requires content-analysis CA with a field of unknown data but of the same length. If the field f has no fixed length, but a terminator constant c is specified in the content-analysis, the content-analysis is removed, and field f is now **`terminatedBy`** c.

Remove Data Dependencies

The second transformation removes data dependencies. All references to values or sizes defined elsewhere in the description are removed. An example of this is the `chunkdata` field as shown on line 21 in Figure 5.1 where **`size`** depends on the value of `length` on line 19. There are two types of data dependencies that are dealt with differently. First, if the contents of a field are defined by reference to another field, the reference is removed by clearing the content specification. The field's value becomes "undefined". The transformation rule implementing this transformation is as follows:

$$f\texttt{: } E[f']\ \texttt{size}\ n \Rightarrow f\texttt{: size } n\texttt{;}$$

If the value of a fixed-length field f is defined by some expression E referencing field f', the value specification is simply removed.

Second, if the size specification of a field depends on another field, the transformation is more involved. It is not possible to clear the size specification of a field just like with value dependencies, since then the position of a following field or structure becomes undefined. Instead, we remove the entire field from its containing structure. To ensure that the generated validator still works, we locate the

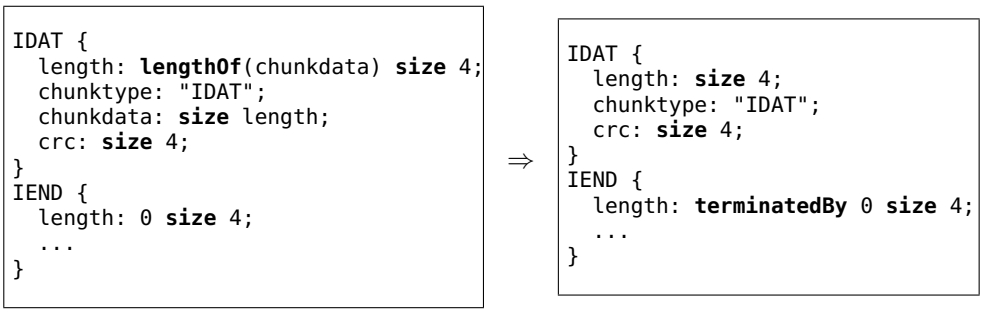

Figure 4.4: Example of *Remove Data Dependencies*.

first field f' that defines a constant value c that is required to follow the removed field f; if s does not define such a field itself, we find the first following structure that does, using the format's sequence. We replace the definition of f' with f'`: terminatedBy` c`;`. To prevent backtracking in the generated validator, we remove any non-mandatory structures (indicated by `*`, `?`, and `()`) inbetween f and f'. To find the first mandatory field that defines a constant, we use a simple algorithm, similar to the computation of first-sets of context-free grammars [ALSU06].

Figure 4.4 shows the effect of a single transformation step to remove the size dependency of the `chunktype` field of PNG's `IDAT` structure[3]. In this example the content-analysis and value dependencies have already been removed. In this step, the `chunkdata` field has been removed from `IDAT`. Additionally, the `length` field of `IEND` has been changed to include the **`terminatedBy`** modifier, because it is the first mandatory constant field following the removed `chunktype` field.

Reduce to Header-Footer Matching

The third and last model transformation reduces a format description to two patterns: one for the beginning and one for the end of the file. This is the same strategy that is employed by the Scalpel carver [RR05]. It requires file formats to have a clearly defined header and footer, using only constants. As a result, a validator based on this description will hardly ever reject data since for every header some footer is very likely to be found (assuming a large amount of files or fragments in the input data). Fragmentation in the input data will lead almost certainly to false positives. However, all recovered files are collected in a single linear pass over the input data.

[3]Note that the `IDAT` structure no longer inherits from the `Chunk` structure; the inheritance hierarchy has been flattened during normalization.

```
sequence
  s e

structures
  s { header: 137, 80, 78, 71, 13, 10, 26, 10; }
  e { footer: terminatedBy 0, 0, 0, 0, "IEND", 0xAE, 0x42, 0x60, 0x82; }
```

Figure 4.5: Example of *Reduce to Header/Footer*.

The transformation operates as follows. Let S be the largest sequence of non-optional consecutive structures starting from the beginning of the sequence definition of the file format. Let E be a similar list of structures, but now starting backwards, from the end of the sequence definition. Now collapse both S and E into single structures s and e by taking the largest sequence of constant fields starting from the beginning and the end respectively, and concatenating consecutive field constants into single constants a and b. Then define the structures s and e as s { `header:` a; } and e { `footer:` **`terminatedBy`** b; }. Finally, construct a new file format with sequence s e. The resulting file format searches for the constant header pattern a, and (if found) subsequently searches for the constant footer pattern b.

Figure 4.5 shows the result of applying this transformation to the full PNG description of Figure 5.1. Note that all consecutive constant fields in the *IEND* structure have been merged into the single field *footer* to construct the largest possible constant.

4.4 Evaluation

To evaluate the effect of the transformations we have applied them on three Derric file format specifications, namely for JPEG, GIF and PNG. We have run the resulting $3 \times 4 = 12$ carver configurations on a representative disk image of 1TB, containing over a million recoverable files. We have then compared the difference in runtime performance, precision and recall between the configurations.

Development of Benchmark Disk Image

The largest publicly available disk image for exercising file carvers is 40GB in size[4]. This, however, is not large enough to properly assess how an application deals with scalability issues in practice. We have therefore developed our own 1TB test set based on data downloaded from Wikipedia. The size of Wikipedia means we could

[4] `http://digitalcorpora.org/corpora/disk-images`

get enough files to fill at least a significant part of the 1TB data set we wanted to create. We used the latest available static dump of all images on Wikipedia, which dates from 2008[5]. Attempting to download all files from that list resulted in around 50% errors due to missing files. The end result was a usable set of over 1.2 million files with a total size of 357GB. An overview of how the files are distributed over each type (JPEG, GIF and PNG) and their total sizes is shown in the first column of Table 4.1.

These files were written into the test image file, spread out across the entire 1TB. Space between files (or fragments) was filled using 543GB of random data and 100GB of only zeros. Although there is little known about the amount and size of zero data blocks on hard drives, we believe 10% is a low estimate, which means the test image is more challenging for file carvers (since zeros are relatively easy to disqualify).

93% of the files have been written into the test image in contiguous blocks and are therefore not fragmented. 3% has been split into two parts and the remaining 4% has been divided into four equal size groups of 3, 4, 5–10 and 11–20 fragments, corresponding to observations of fragmentation in the wild [Gar07]. Splitting was done at random locations in the files, but always on a cluster boundary of 4096 bytes, corresponding to the smallest common cluster size.

Format	Configuration	Running time	True positives	False positives	Precision	Recall
JPEG	Base	742m	882,511	0	100.0%	94.9%
input data:	NoCA	295m	860,022	22,007	97.5%	92.4%
files: 930,424	NoDD	231m	837,382	46,561	94.7%	90.0%
size: 327GB	Header	231m	837,382	46,561	94.7%	90.0%
GIF	Base	320m	34,078	0	100.0%	93.2%
input data:	NoCA	267m	33,210	702	97.9%	90.8%
files: 36,576	NoDD	231m	32,912	2,780	92.2%	90.0%
size: 3GB	Header	231m	32,912	2,780	92.2%	90.0%
PNG	Base	691m	222,660	0	100.0%	94.2%
input data:	NoCA	280m	219,001	8,073	96.4%	92.6%
files: 236,457	NoDD	231m	212,911	13,905	93.9%	90.0%
size: 27GB	Header	231m	211,790	14,577	93.6%	89.6%

Table 4.1: Results per configuration for all three file formats.

[5] `http://static.wikipedia.org/downloads/2008-06/en/images.lst`

Execution of the Benchmark

The 12 carver configurations have been run on a 3.4GHz Intel Core i7-2600 with 8GB of RAM and an attached 2TB 10.000RPM SATA harddrive. The operating system used was Ubuntu Linux 11.04, with Oracle's JDK 1.6.0 update 13. The results of each run are shown in Table 4.1. For each file type and configuration it shows the wall clock running time in minutes in the third column. The fourth and fifth column of each table display the number of true and false positives respectively. True positive means a file has been recovered that was actually present in the disk image. False positive means that the file carver recovered a file erroneously, for instance, by combining a file header with the wrong footer. The last two columns give precision and recall percentages.

Analysis of Results

The fastest two configurations, NoDD and Header, require the same amount of time to complete for each format. The 231m corresponds to the time required to read through a terabyte of data on the hardware used, indicating that when using the NoDD and Header configurations, the application is bound by the read performance of the underlying platform. In other words, reading all data in a single linear pass would take the same amount of time.

Additionally, on JPEG and GIF, both the NoDD and Header configurations return exactly the same results, indicating that the final transformation does not impact the quality of the results or runtime performance. However, on PNG the situation is different: the NoDD configuration returns a little more true positives and fewer false positives.

This difference can be traced to the fact that the descriptions for JPEG and GIF both have a large variable block in the middle that is effectively eliminated by the *remove data dependencies* transformation, while the PNG description does have a fixed structure at a variable location between the first and final structure (the *IDAT* structure). This causes the PNG NoDD configuration to be more discriminating than the Header configuration. The result is slightly higher precision and recall.

For all three formats, the Base configuration returns no false positives, reaching 100% precision. The Base descriptions are complete, which leads to validation of all the contents of a candidate match. Since all three formats are compressed, even a single missing or misplaced fragment will lead to errors during validation and be rejected by the validator.

Another point of interest is the running time of the Base configuration. For JPEG and PNG, this is both at least twice the time required to run the NoCA configuration and at least three times the amount of time required to run the NoDD and Header configurations. Two factors contribute to this. The first factor is the rela-

tively expensive operations by the validators. An example of this is CRC calculation. Although an optimized implementation is used, due to fragmentation, the CRC is sometimes calculated over large blocks that end up not being matches.

The second factor is the effect of fragment reordering in Excavator. Whenever a validator rejects a candidate match, an additional check is performed to determine whether a possible footer of the same file format is relatively close to the error location. If this is the case, the clusters between the error location and the matching footer are partially reordered and removed, running the validator on each combination to determine possible hits. To prevent a combinatorial explosion, reordering is only enabled when the distance between error location and footer is smaller than 2MB. Consequently, it is triggered by the most precise validators. In the more permissive validators the gap size is either too large or it is entirely undetected (and leads to a false positive in the results).

4.5 Discussion

Effects on Analysis Time

It can be argued that, although more permissive validators will run faster, in practice, they may end up requiring more of the investigator's time, because there are more false positives to inspect. This time could also be spent running the analysis using a higher precision validator. Depending on the legal case, however, it might be more valuable to have results more quickly: even with more false positives, a crucial piece of evidence could be found earlier.

With our current results we believe the transformed validators are a useful alternative to the most precise validators, since the loss of precision and recall (8% and 5% respectively) is relatively small compared to the gain in performance (between 40% and 320%). For example, for PNG, the fastest carver returns 211,790 true positives and 14,577 false positives but it requires only 1/3rd of the running time of the most precise carver.

At the same time, the fastest validators do not make the original validators obsolete, considering that, after the fastest validator has finished, the most precise JPEG validator is able to recover 45,129 true positives in the extra 510 minutes.

An alternative approach is to use the more precise validators for only a short period of time and use their intermediate results when time runs out. While this is possible, there is a chance that the more precise validator will spend a lot of time near the beginning of the disk image recovering a fragmented file, while the fastest validator (which does not reject anything) will skip over it and return all the relatively simple matches directly.

Another alternative approach is to use one of the fastest validators and run the most precise validator on the results to remove false positives. This may help all

carver configurations achieve 100% precision.

Other File Formats

Our experiment takes three popular image file formats and shows how the described model transformations affect runtime performance and precision of the generated validators from their descriptions. A question is whether this approach works as well on other file formats. There is a strong indication that they will perform similarly, considering that most forensically interesting file formats tend to either be multimedia, document or container files. All three of these types of files often have features comparable to the image file types we used: extensive metadata, compressed contents and well-defined headers and footers. Examples of forensically interesting file types that are structured similarly are AVI and MPEG for multimedia, XLS and PDF for documents, and ZIP and RAR for containers. In future work we will apply Excavator and the model transformations on Derric descriptions of these file formats.

4.6 Related Work

Transformation for optimization is as old as compiler construction [AC72]. Moreover, transformation is considered to be one of the cornerstones of model-driven engineering [Sch06, Bézo6] and generative programming [CE00]. In both areas the objective is to specify the essential variability of an application domain at high levels of abstraction, and then generating the low-level code automatically. The commonality of an application domain is captured by such transformations. We have applied this well-known pattern in the context of digital forensics.

Domain-specific analysis, verification, optimization, parallelization and transformation (AVOPT) are well-known reasons for DSL development [MHS05]. In particular, for optimization, the explicit representation of high-level domain concepts can be used by a compiler in order to generate code that is more efficient. Such optimizations are very hard to obtain in the context of ordinary, hand-written programs, since the high-level domain concepts are lost in low-level code. In this chapter we have shown how to use domain concepts of Derric (content analysis, data dependencies and header/footer) in order to obtain faster file carvers.

In [CBDM01] the authors present a model and strategy for transforming source code in order to reduce the energy consumption of a program. It includes an explicit cost model of both the transformations and the object program. Our transformations themselves are very inexpensive, and the cost model for file carving is based solely on the most expensive operations at runtime. Another instance of applying model transformation for optimization is presented in [BJS10]. The authors apply a number of successive transformations on BIP (Behavior, Interaction, Prior-

ities) models to obtain a single monolithic, efficient program. The DERRIC model transformations operate in the same way in that they remove overhead elements from the input model. What makes our transformations different from such approaches, however, is that the transformations are not (strictly) semantics preserving, as they discard information. As such the transformations can be considered approximations, in a similar way that context-free grammars can be approximated by regular expressions [MN00].

Our software tool EXCAVATOR represents the state-of-the-art in digital forensics data recovery, implementing fragmented file recovery [Gar07, Coh07] and a stream-based processing model [Gar10]. Furthermore, our model-driven approach distinguishes itself by allowing high-level specification of elaborate data structures not implemented in popular file carvers. By comparison, PHOTOREC [Gre09] requires hand-written format validators and SCALPEL [RR05] employs regular expressions for format validation.

4.7 Conclusion

Modifiability, runtime performance and scalability are the major challenges in digital forensics software construction. Moreover, forensic investigations are often constrained by very strict deadlines. As a result digital forensics software is often modified on a case-by-case basis. This just-in-time "carver hacking" is error prone and time consuming.

In previous work we have introduced a model-driven approach to digital forensics software development, DERRIC, which improves performance and modifiability by generating efficient code from high-level file format descriptions. In this chapter we introduced three source-to-source model transformations on DERRIC descriptions in order to make the trade-off between precision and runtime performance configurable. This allows investigators to choose performance over precision if time constraints should require so, or the other way around,—without having to change any code.

The effect of the model transformations is evaluated on a 1TB disk image containing over a million recoverable files, specifically constructed to resemble a realistic file carving scenario. Our results show that performance gains up to a factor of three can be achieved. This comes at a loss of up to 8% in precision and 5% in recall.

Part III

Maintainability

Highlights:

- Evaluation of the maintenance characteristics of Derric.
- Design and implementation of the Trinity interpreter-based Derric IDE.

CHAPTER 5

A Case Study in Evidence-Based DSL Evolution

This chapter was previously published as a paper with the same title in *9th European Conference on Modelling Foundations and Applications (ECMFA'13)* [BS13a]. Joint work with Tijs van der Storm.

Abstract

Domain-specific languages (DSLs) can significantly increase productivity and quality in software construction. However, even DSL programs need to evolve to accomodate changing requirements and circumstances. How can we know if the design of a DSL supports the relevant evolution scenarios on its programs? We present an experimental approach to evaluate the evolutionary capabilities of a DSL and apply it on a DSL for digital forensics, called Derric. Our results indicate that the majority of required changes to Derric programs are easily expressed. However, some scenarios suggest that the DSL design can be improved to prevent future maintenance problems. Our experimental approach can be considered first steps towards evidence-based DSL evolution.

5.1 Introduction

Domain-specific languages (DSLs) can increase productivity by trading generality for expressive power [MHS05, DKV00]. Furthermore, DSLs have the potential to improve the practice of software maintenance: routine changes are easily expressed. More substantial changes, however, might require the DSL itself to be changed [DK98]. How can we find out whether the relevant maintenance scenarios will require routine changes or not?

In this chapter we present a test-based experimental approach to answer this question and apply it to a domain-specific language for describing file formats: Derric (see Chapter 3). Derric is used in the domain of digital forensics to generate software to analyze, reconstruct, and recover file-based evidence from storage devices. In digital forensics it is common that such file format descriptions need to be changed regularly, either to accomodate new file format versions, or to deal with vendor idiosyncrasies.

As a starting point, we have assembled a large corpus of image files to trigger failing executions of the file recognition code that is generated from Derric descriptions. Each failing execution is attempted to be corrected through a modification of the Derric code, until all image files are correctly recognized. The required changes are accurately tracked, categorized and rated in terms of complexity. This set of changes provides an empirical baseline to assess whether the design of Derric sufficiently facilitates necessary maintenance.

The results show that all of the required changes were expressible in Derric; the DSL did not have to be changed to resolve all failures. The majority of harvested changes consists of multiple, inter-dependent modifications. The second most common change consists of a single, simple, local modification. Finally, a minority of changes is more complex. We discuss how the Derric DSL may be changed to make these changes expressed more easily. Thus, the experiment has provided us with empirical data to improve the design of Derric.

The contributions of this chapter can be summarized as follows:

- We describe and apply an experiment in DSL-based maintenance in the context of Derric, and provide a detailed description including its parameters.
- We present empirical results on how the Derric DSL supports the maintenance process in the domain of digital forensics.
- We discuss the usefulness of this approach and how it has helped us to both evaluate and improve the design of Derric.

These contributions can be considered first steps towards evidence-based DSL evolution.

```
format PNG
  strings ascii
  sign false
  unit byte
  size 1
  type integer

sequence
  Signature IHDR
  Chunk* IDAT IDAT* Chunk*
  IEND

structures
Signature {
  marker: 137,80,78,71,13,10,26,10;
}

Chunk {
  length: lengthOf(chunkdata) size 4;
  chunktype: !"IDAT" size 4;
  chunkdata: size length;
  crc: checksum(
        algorithm="crc32-ieee",
        start="lsb",store="msbfirst",
        fields=chunktype+chunkdata)
       size 4;
}

IHDR = Chunk {
  chunktype: "IHDR";
  chunkdata: {
    width: !0 size 4;
    height: !0 size 4;
    bitdepth: 1|2|4|8|16;
    colourtype: 0|2|3|4|6;
    compression: 0;
    filter: 0;
    interlace: 0|1;
  }
}

IDAT = Chunk {
  chunktype: "IDAT";
  chunkdata: compressed(
               algorithm="deflate",
               layout="zlib",
               fields=chunkdata)
             size length;
}

IEND {
  length: 0 size 4;
  chunktype: "IEND";
  crc: 0xAE, 0x42, 0x60, 0x82;
}
```

Figure 5.1: Simplified PNG in DERRIC.

5.2 Background

DERRIC is a DSL to describe binary file formats (see Chapter 3). It is used in digital forensics investigations to construct highly flexible and high performance recovery tools. One example is the construction of file carvers (see Chapter 2), which are used to recover possibly damaged evidence from confiscated storage devices (e.g., hard disks, cameras, mobile phones etc.). DERRIC descriptions are used to generate some of the software components, called *validators*, that check whether a recovered piece of data is a valid file of a certain type.

An example DERRIC description for a simplified version of the PNG file format is shown in Fig. 5.1. The structure of a file format is declared using the **`sequence`** keyword. The sequence consists of a regular expression that specifies the syntax of a file format in terms of basic blocks, called *structures*. In this case, a PNG file starts with a `Signature` block, an `IHDR` block, zero-or-more `Chunks` and finally an `IEND` block.

The contents of each structure is defined in the following **`structures`** section.

A structure consists of one or more fields. The contents and size of each field are constrained by expressions. The simplest expression is a constant, that directly specifies the content, and hence length, of a field. This is the case for the `marker` field of the `Signature` structure. Another common type of constraint only restricts the type and/or length of a field. For instance, the `chunktype` field of structure `Chunk` is constrained to be of type `string` and size `4`. Constraints may involve arbitrary content analyses. For example, consider the `crc` field. To recognize this field a full checksum analysis following the `crc32-ieee` algorithm should be performed.

5.3 Observing Corrective Maintenance

To study the maintainability characteristics of Derric, we need a way to inspect and evaluate actual maintenance scenarios. In other words: we need to observe how DSL programs are changed. For the purpose of this chapter, we focus on *corrective* maintenance [ISO06], which is maintenance in response to observed failures ("bug fixing").

To realize this, a large corpus of representative and relevant inputs to a DSL program is needed, which allows us to automatically generate failures, which in turn trigger corrective maintenance actions. The approach is similar to *fuzzing* where a program is run on large quantities of invalid, unexpected or even random input data [Oeh05]. For maintenance evaluation, however, it is of paramount importance that the data is representative of what would be encountered in practice.

In the case of Derric we have assembled a large, representative corpus of image files (JPEG, GIF and PNG) for which Derric descriptions are available. The exact nature of these descriptions and the corpus is described in detail in Section 5.4.

For each file format f, the initial Derric D_f^i description is compiled to a validator and subsequently run on the corpus files of type f. This results in an initial set of files for which validation fails[1]. The set of failures is then divided over equivalence classes which are sorted by their size. This allows us to focus on the most urgent problems first. Next, D_f^i is edited to obtain a new version D_f^{i+1} which resolves at least one of the failures in the largest equivalence class. As soon as the set of failures is observed to decrease, D_f^{i+1} is committed to the version control system. Before committing we ensure that the set of correctly validated files (the true positives) strictly increases, as a form of regression test. The process then repeats, now using D_f^{i+1} as a starting point.

After all failures have been resolved, the changes, as stored in the version control, are categorized in *change complexity classes*. A change may thus be interpreted as being more complex than another change. This provides an empirical base line

[1] Technically, both false positives and false negatives are failures. However, since the corpus only contains real files, we cannot detect when a validator would incorrectly validate a file.

to qualitatively assess to what extent Derric supports maintenance of format descriptions.

5.4 Experiment

DSL Programs and Corpus

The three DSL programs that have been used are Derric descriptions of JPEG, GIF and PNG. These file formats are well-known, very common and highly relevant to the practice of digital forensics. An impression of the sizes of these descriptions is given in Table 5.1. From the table it can be inferred that the descriptions are significantly different. Both GIF and PNG have a richer syntactic structure than JPEG. Structure inheritance is heavily used in JPEG and PNG but only once in GIF. Finally, GIF has a lot more fields per structure (58 per 12). Summarizing, we claim that the three file format descriptions cover a wide range of Derric's language features, in different ways.

	JPEG	GIF	PNG
Sequence tokens	14	29	30
Structures	15	12	20
Uses of inheritance	10	1	17
Field definitions	32	58	27

Table 5.1: Initial Derric descriptions.

	Data Set		Failures	
Format	#	size	#	%
JPEG	930,386	327GB	5,485	0.6%
GIF	36,524	3GB	389	1.1%
PNG	236,398	27GB	5,789	2.4%
Total	1,203,308	357GB	11,663	1.0%

Table 5.2: Initial validator results.

The second important component of the experiment, is a representative corpus. We have developed such a corpus for the evaluation of our earlier work on model-transformation of Derric descriptions (see Chapter 4). This data set contains JPEG, GIF and PNG images found on Wikipedia, downloaded using the latest available

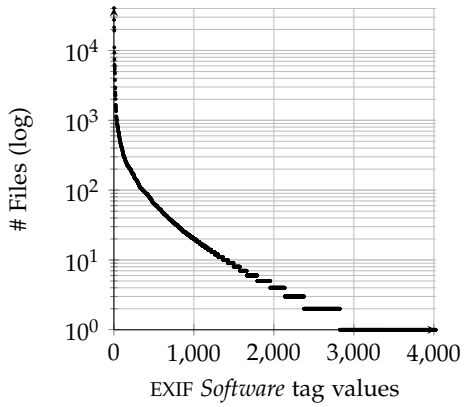

Figure 5.2: Distribution of EXIF *Software* tag values over 28.4% of the corpus.

static dump list, which dates from 2008[2]. Around 50% of the files on that list were still available and included in the set. An overview of the data set is shown in Table 5.2. The corpus contains a total of 1,203,410 images, leading to a total size of 357 GB. As the last two columns show, not all images in the data set are recognized by the validators generated from the respective JPEG, GIF and PNG descriptions: between 0.6% and 2.4% of the files in the data set are not recognized using the base descriptions of the respective file formats.

The Wikipedia data set can be considered representative, since the files uploaded to it originate from many different sources (e.g., cameras, editing software, etc.). We have verified this diversity by inspecting the metadata of the files and aggregating the results.

This shows that the set contains files from a large number of different cameras (e.g., Canon, Nikon, etc.) Furthermore, many images have been modified using a multiplicity of tools (e.g., Photoshop, Gimp, etc.) Original computer images such as diagrams and logos have been created using many different tools (e.g., Dot, Paintshop Pro, etc.)

The diversity is depicted graphically in Fig. 5.2, showing the distribution of files over values of the EXIF *Software* tag present in 28.4% of the images. The most common tool is Photoshop 7.0, used on 3.4% of the corpus; Photoshop CS2 and CS (Windows) are used on 2.3% and 1.8% respectively. ImageReady covers 1.6%. After that the percentages rapidly decrease: no specific version of any application was used in more than 1% of the files. The number of different values is 4,024.

[2]Available at `https://github.com/jvdb/derric-eval`

	Structures	Sequence
Add	Add new structure	Insert structure symbol
Modify	Add, modify, or delete field	Change regular grammar
Delete	Remove structure definition	Remove structure symbol

Table 5.3: Edit semantics: a DERRIC description's two main sections can be edited in three ways.

Classifying and Ordering Failures

To improve productivity and handle the most relevant issues first, the set of failures is divided over equivalence classes, according to their *longest normalized recognized prefix*: this is the sequence of DERRIC structures that has been successfully recognized before recognition failed. Classification is repeated after each iteration, because after each change to a description, files might now fail with another prefix.

The prefix is normalized to eliminate the common effect of repeating structures. For instance, if the recognized prefix consists of the structures `A B B C`, then the normalized prefix is `A B+ C`. The plus-sign indicates one-or-more occurrences. As a result, files that failed recognition with prefixes `A B C`, `A B B C`, `A B B B C`, etc. all end up in the same bucket. The equivalence classes thus obtained are then sorted according to size in order to first improve those parts of the description that generate the most failures.

Evolving the Descriptions

The next step in the experiment is to manually fix the descriptions until all failures have been resolved. After each change, we recorded how many *edits*—additions, modifications and deletions—were needed to reduce the number of failures. An edit captures an atomic delta to a description. Edits can be applied to either the sequence or the list of structures. The semantics of edits is summarized in Table 5.3.

The simplest edits are addition/removal of a structure to/from the structures section of a DERRIC description, and adding/removing a referenced structure from the sequence expression (cf. Fig. 5.1). Furthermore, a structure itself can be modified by adding, modifying or removing fields. The sequence can be modified by changing the regular expression without adding or removing a structure reference.

Each change has been tracked in the Git version control system[3] to allow full traceability and reproducability of the results of this chapter. In fact, a single change corresponds to a single commit. After each change the DERRIC compiler was rerun

[3]Available at `https://github.com/jvdb/derric-eval`

with the modified descriptions. The process was repeated until all failures were resolved.

Change Complexity Classes

After all failures have been resolved, the resulting set of changes is divided over equivalence classes according to their *change complexity*. Change complexity is intuitively defined in terms of the number of edits in a change, their interrelatedness and how much they are scattered across a source file: more edits, more interrelatedness and more scattering, means higher complexity.

A change consisting of a single edit has very low change complexity. On the other hand, a change involving many logically related edits, scattered over the whole program, has a high change complexity. Simple, low complexity changes leave the structure of the original program mostly intact. At the opposite end, high complexity changes might well create future maintenance problems.

Just like code smells [FBB+99] might be indicators of software design problems, in the case of Derric, we conjecture, high complexity changes might indicate *language design problems*. For the purpose of our experiment we have identified 3 change complexity classes. Below we briefly describe each class, rated as *Low*, *Medium* or *High*.

- *Single, localized edit (Low)* The ideal situation is where a change requires a single modification of the program. By implication, such a change is always localized. Example: a single edit of the sequence, or the change of a single field in a structure.
- *Multiple, but dependent edits (Medium)* In this case, a change requires multiple, inter-dependent edits. For instance, defining a new structure, then adding a reference to it in the sequence section.
- *Cross-cutting changes (High)* Cross-cutting changes require many (more than two) similar edits scattered across the program. Such changes always involve some form of duplication. This kind of changes is very bad, since they affect the program in a way that is dependent on the size of the program.

The changes, categorized in the change complexity classes, provide an empirical base line to start discussing to what extent Derric supports maintenance.

5.5 Results

The results of the experiment are summarized in Table 5.4, 5.5 and 5.6 for the file formats JPEG, GIF and PNG respectively. The first column of each table identifies

the change (i.e., set of edits). In the following, we will identify changes by using a combination of file format name and Id, like so: PNG 11 denotes the eleventh change of the PNG description in Table 5.6. Columns 2-5 display how many edits of that particular type were required in order to decrease the number of failures. For instance, change JPEG 1 involved two edits: a structure definition was added, and a reference was added to the sequence expression. Note that deletions are omitted from these tables since they never occurred.

The actual decrease in failures is shown in the "Errors Resolved" column. Finally, the last column shows how a change was categorized with respect to change complexity. Revisiting change JPEG 1 we see that it is ranked as *Medium*, which means that the change contains multiple, dependent edits. Hence we can conclude that the reference inserted into the sequence expression has to be a reference to the newly added structure.

	Structure		Sequence		Errors	
Id	Add	Mod	Add	Mod	Resolved	CC
1	1		1		520	*Medium*
2			1		284	*Low*
3	1		1		245	*Medium*
4	1		1		821	*Medium*
5				1	3395	*Low*
6				1	138	*Low*
7	1		2		46	*High*
8	1	4	21		26	*High*
9	1		4		5	*High*
10	1		19		3	*High*
11	1		2		2	*High*

Table 5.4: Modifications to the JPEG description.

5.6 Analysis

To summarize the results of our experiment, Table 5.7 shows the total number of changes per complexity level. The table shows that the majority of changes are easily supported by Derric: 13 are simple, localized edits (*Low*), and 19 changes require multiple, dependent edits. The dependency between edits in these changes is a direct consequence of separating sequence from structure definition. In other words: this dependency is anticipated by the design, and hence unavoidable.

Only 5 changes are categorized as cross-cutting (*High*). While in the experiment these changes did not occur very frequently, they still might indicate there is room

	Structure		Sequence		Errors	
Id	Add	Mod	Add	Mod	Resolved	CC
1		1			9	*Low*
2			1		115	*Low*
3			1		137	*Low*
4		3			36	*Medium*
5			1		39	*Low*
6				1	48	*Low*
7				1	3	*Low*
8			2		2	*Medium*

Table 5.5: Modifications to the GIF description.

for improving the design of Derric. Moreover, looking at the results for JPEG, we seem to observe a pattern of deterioration. Investigating the actual changes reveals that, indeed, duplication introduced by earlier changes, has a detrimental effect on the required subsequent changes. The fact that cross-cutting changes may amplify each other, is exactly the evolutionary effect we would like to avoid. Three language features could be introduced to Derric to eliminate such cross-cutting changes completely:

- Abstraction: a language construct to declare subsequences so that duplicate subsequences can be referred to by name.
- Padding: a construct to automatically interleave certain bytes inbetween structure references in the sequence declaration.
- Precedence: declaring that a particular structure has priority over another one.

Below we motivate these language features based on the results of the experiment.

Abstraction In JPEG 7, a newly discovered data structure `SOF1` is added to the description. It was discovered that it is part of a sub-sequence of structures that may occur both before and after a mandatory `SOS` structure. As a result, a reference to `SOF1` had to be inserted in two places. The relevant part of the original sequence reads as follows:

```
sequence ...
(DQT DHT DRI SOF0 SOF2 APPX COM)*
SOS
(SOS DQT DHT DRI SOF0 SOF2 APPX COM)*
```

	Structure		Sequence		Errors	
Id	Add	Mod	Add	Mod	Resolved	CC
1	5		5		3136	*Medium*
2	1		1		1819	*Medium*
3	1		1		332	*Medium*
4	1		1		63	*Medium*
5	1		1		73	*Medium*
6	2		2		112	*Medium*
7	1		1		144	*Medium*
8	1		1		24	*Medium*
9			1		20	*Low*
10			1		18	*Low*
11				1	20	*Low*
12	1		1		10	*Medium*
13	1		1		2	*Medium*
14	1		1		9	*Medium*
15	2		2		2	*Medium*
16			1		3	*Low*
17	1		1		1	*Medium*
18			3		1	*Medium*

Table 5.6: Modifications to the PNG description.

Level	Name	#
Low	Single localized	13
Medium	Multiple dependent	19
High	Cross-cutting	5
	Total	37

Table 5.7: Changes per change complexity class.

Note that the sequence `DQT DHT DRI SOF0 SOF2 APPX COM` is duplicated. An abstraction construct would allow the description to be refactored as follows:

```
def Seq = DQT DHT DRI SOF0 SOF2 APPX COM;
sequence ... Seq* SOS (SOS Seq)*
```

To accomodate the new `SOF1` structure, only the definition of `Seq` would have to be adapted. Such an abstraction mechanism feature would not only reduce the severity of such changes, it would also clearly communicate to readers of the description that the sequences before and after the SOS reference are always the same.

Padding The JPEG 8 change clearly signals a problem: padding bytes are allowed everywhere in between structures. Every change that modifies the sequence will explicitly make sure that padding is maintained. The duplication introduced by JPEG 7 makes the way this change is expressed even less desirable. A (domain-specific) padding construct allows padding to be expressed in a single place in the configuration section:

```
padding 0xFF
```

The compiler would then weave the generic padding element into the sequence.

Precedence The cross-cutting change JPEG 10 signals another language feature that could be added to Derric. A new structure `COMElanGmk` was identified, which functions as an alternative to the standard `COM` structure. The only difference from `COM` is that `COMElanGmk` redefines the contents of a single field using Derric's support for structure inheritance. We would, however, like to also express that `COMElanGmk` has precedence over `COM`: if it is there, consume it, *otherwise* attempt to match `COM`.

The current resolution involves duplicating large parts of the sequence to move the choice between either structure to a higher level. A proper solution would be to extend the set of sequence operators (?, *, etc.) with a new binary operator $<$. The precedence ordering could then be expressed simply as `COMElanGmk < COM`.

5.7 Discussion

Lessons Learned

Based on this case study, we can draw a number of conclusions that are generally applicable to the area of DSL development and model-driven development at large. First of all, in order to do evidence-based DSL evolution, the existence of a large, representative corpus is of paramount importance. Given such a corpus, it becomes possible to apply our test-based experimental approach. Our results show that such an experiment indeed provides useful feedback on the design of a DSL.

The corpus of files used in our experiment in essence represents a very large and comprehensive test suite. In other domains, such a test suite has to be designed up front. Nevertheless, the existence of test suites for (legacy) code, could thus be instrumental in deciding whether to adopt a model-driven approach. For instance, in [LSVW10] the authors perform a study whether the Mod4J framework is suitable to build web applications following a reference architecture. In this case, the organization had ample experience building such web applications. If (evolving) test suites for a representative sample of non-Mod4J applications exist, they can be run

against Mod4J replicas to find out whether Mod4J supports the necessary evolution facilities to fix the failing tests.

Second, to our surprise, the experiment showed that even a simple DSL such as DERRIC requires abstraction facilities in order to mitigate future maintenance. Maybe DSLs and modeling languages are much more like programming languages than we might think. As such, our results provide a cautionary tale, which may be taken into consideration when designing a DSL or modeling language. Furthermore, it might suggest that, if such a feature is to be avoided, that graph-like, visual concrete syntax is preferrable, since it would allow the direct representation of sharing of sub-structures.

Finally, since our experiment requires the accurate tracking and classification of changes to source models, textual syntax seems to be an advantage. The textual syntax of DERRIC allowed us to use standard `diff` tools to get insight into what was changed inbetween revisions. A visual modeling language would most certainly require custom, domain-specific difference algorithms [XS05]. Generic difference algorithms (on trees or graphs) would likely contain irrelevant noise, and hence would be hard to interpret.

Threats to Validity

Even though our classification of changes is informal, we contend that it is sufficiently intuitive. Proficient users of computer languages (domain-specific or general purpose) use similar reasoning to distinguish "good" changes from "bad" changes. Most programmers are familiar with the principles of Don't-Repeat-Yourself (DRY) and Once-and-Only-Once (OAOO). These are precisely the principles that were violated in the cross-cutting changes.

The changes were performed by the first author (the designer of DERRIC) who has ample experience in digital forensics. As such, he could have tended towards the smallest and simplest changes. However, in order to evaluate the way a language supports maintenance it is essential to analyze *optimal* changes; only then can the language aspect be isolated. A subject who is less versed in the domain of digital forensics or DERRIC, would probably have added noise to the results (i.e., unneeded complexity in the changes), and consequently, the results would have been harder to interpret.

As shown in Section 5.4, we consider the set of image files from Wikipedia a suitable test set for generating failures and harvesting changes. First, the set of images is constructed by thousands of users of Wikipedia, so there is no selection bias. Second, there is a high variability in the origin of the images and how the images were processed in user programs (Fig. 5.2). Finally, the data set is large enough to generate realistic failures; any of the observed failures could have occurred in practice.

It could be argued that neither JPEG, GIF nor PNG are rich enough to cover the full expressivity or expose the lack thereof of Derric. This might be true, however, the Derric language is designed precisely for this kind of file formats. In Section 5.4 we have argued that the Derric descriptions of these file formats are sufficiently different to cover the whole language.

Related Work

Mens et al. [ME05] define evolution complexity as the computational complexity of a metaprogram that performs a maintenance task, given a "shift" in requirements. Our classification of changes is comparable since we consider small and local edits (fewer "steps") to be easier than multiple, dependent and scattered edits (requiring more steps). Making this relation more precise, however, is an interesting direction for further research. This would involve formalizing each change as a small metaprogram, and then using its computational complexity to rank the changes.

Hills et al. [HKSV11a] do a similar experiment but use an imaginary virtual machine for "running" maintenance scenarios encoded as simple process expressions. Since the changes and programs investigated in this chapter are relatively small, writing them as *actual* metaprograms might be practically feasible. Even more so since Derric is implemented using the metaprogramming language Rascal [KSV09a], which is highly suitable for expressing the changes as source-to-source transformations.

The work presented in this chapter can be positioned as an experiment in language evaluation. Empirical language evaluation is relatively new since, as pointed out by Markstrum [Mar10], most language features are introduced without evidence to back up its effectiveness or usefulness. In the area of DSL engineering, however, there is work on evaluating the effectiveness of DSLs with respect to program understanding [MHS05], key success factors [HPD09], and maintainability [KSV10]. Our experiment can be seen in this line of work, but focusing on how a DSL *as a language* supports evolution.

Corpus-based language analysis dates at least from the '70s, but is getting more attention recently; see [FGLP10] for a comprehensive list of references. A recent study is performed by Lämmel and Pek. [LP10]. The authors have collected over 3,000 privacy policies expressed in the P3P language in order to discover how the language is used and which features are used most. Morandat et al. [MHOV12] gather a corpus of over 1,000 programs written in R to evaluate some of the design choices in its implementation. A difference with respect to our work, however, is that corpus-based language analysis focuses on a corpus of *source files*. Instead, in this chapter we used a corpus of *input files* to trigger realistic failures, *not* to analyze the usage of language features, but to analyze how these features fare in the face of evolution.

Since the changes we propose for Derric may have an impact on existing descriptions, another perspective on the work in this chapter is that of coupled evolution [DRIP12], In the context of the classification described by Gruschko et al. [GKP07], the changes we propose are all "Not Breaking Changes".

However, it is imaginable that changes that would break existing descriptions could be proposed, so it may be useful to develop a mapping between our classification and the difficulty in automatically migrating Derric descriptions. The feasibility and complexity of automatic migration using tools such as Cope [HBJ09] or Flock [RKPP10] may be a useful metric, although the user's perspective remains the most important for an end-user DSL.

5.8 Conclusion

DSLs can greatly increase productivity and quality in software construction. They are designed so that the common maintenance scenarios are easy to execute. Nevertheless, there might be changes that are impossible or hard to express. In this chapter we have presented an empirical experiment to discover whether Derric, a DSL for describing file formats, supports the relevant corrective maintenance scenarios.

We have run three Derric descriptions of image formats on a large and representative set of image files. When file recognition failed, the descriptions were fixed. This process was repeated until no more failures were observed. The required changes, as recorded in version control, were categorized and rated according to their complexity.

Based on the results we have identified to what extent Derric supports maintenance of file format descriptions. The results show that most of the changes are easily expressed. However, the results also show there is room for improvement: three features should be added to the language. The most important of those is a mechanism for abstraction to factor out commonality in Derric syntax definitions.

Our experimental approach can be applied in the context of other DSLs. The only requirement is a representative corpus of inputs that will trigger realistic failures in the execution of DSL programs and a way to classify and rank the changes required to resolve the failures. By fixing the DSL programs, tracking and ranking the required changes, it becomes possible to observe how seamless (or painful) actual maintenance would be. We consider the experiment presented in this chapter as a first step towards evidence-based DSL evolution.

CHAPTER 6

Trinity: An IDE for The Matrix

This chapter was previously published as a tool paper with the same title in *29th IEEE International Conference on Software Maintenance (ICSM'13)* [BS13b]. Joint work with Tijs van der Storm.

Abstract

Digital forensics software often has to be changed to cope with new variants and versions of file formats. Developers reverse engineer the actual files, and then change the source code of the analysis tools. This process is error-prone and time consuming because the relation between the newly encountered data and how the source code must be changed is implicit. Trinity is an integrated debugging environment which makes this relation explicit using the Derric DSL for describing file formats. Trinity consists of three simultaneous views: 1) the runtime state of an analysis, 2) a hexview of the actual data, and 3) the file format description. Cross-view traceability links allow developers to better understand how the file format description should be modified. Trinity aims to make the process of adapting digital forensics software more effective and efficient.

6.1 Background

Maintenance Challenges in Digital Forensics

The storage capacity of digital devices continues to grow. Forensic software is currently required to analyze data in the terabyte range in very short time frames. This requires perfective maintenance to optimize and tune analysis tools. At the same time, corrective maintenance has to be performed when new variants and versions of file formats are encountered. Most of these variants are non-standard, so standards documents cannot be consulted for the required changes. Moreover, the data is often created by proprietary firmware (e.g., of digital cameras) or other types of closed-source applications (e.g., word processors, photo-editing software). As a result, the source code is generally unavailable for inspection.

Corrective maintenance then boils down to reverse engineering the file format variant based on the binary data itself. This process is quite cumbersome, since the structure of the data is not a first class citizen in general purpose programming languages. In hand-coded file processing software, the layout of a binary file format like PNG [W3C03], for instance, is encoded in complex control-flow and (interdependent) data structures. This means that debugging requires ad hoc decoding of values, inspection of input data to check dependencies between values and manually tracking structural layout and ordering.

Besides time consuming, these steps also tend to be error-prone. For example, an off-by-one error in an offset calculation causes a wrong value to be used, but also shifts interpretation of all consecutive values and their dependencies. Such small errors are hard to catch since there are no explicit links between the input data and how the code interprets it.

When adapting existing implementations of file processing software, interactive debuggers can be used, but they are agnostic to the domain-specific aspects of file formats. Furthermore, each file format may have its own conventions such as whether length fields include or exclude marker values, and whether indices are 0- or 1-based. As a result, reverse engineers have to mentally translate the information that is presented to them.

Trinity is an IDE for reverse engineering binary data which automates a significant portion of this translation. By maintaining semantic links between data, runtime state and code, it becomes possible to *debug the data*, instead of just the code. The key enabler for this is representing file format structure at a higher level of abstraction. Derric is a domain-specific language (DSL) that precisely does that (see Chapter 3.

Declarative File Format Descriptions

DERRIC is a domain-specific language to declaratively describe binary file formats. It allows the definition of the components of a file format (called "structures"), their sequential arrangement, and the possible dependencies between elements. For instance, a file format description may contain structure definitions for headers, footers and data blocks. These structures are arranged sequentially according to a (regular) grammar, capturing the layout of a file format. An example of a dependency is when the length of a certain sequence of bytes is constrained by the value of certain bytes elsewhere in the file. DERRIC provides a configurable language for expressing these and other aspects of file formats.

A DERRIC description is divided in two main sections. The first part of a DERRIC description is the sequence section, which consists of a regular expression capturing the sequential layout of a file format. For instance, the following example presents an abridged version of the layout of PNG (where ellipses indicate omitted details):

```
sequence
  Signature IHDR
    (...)* PLTE? (...)* IDAT IDAT* (...)*
  bBPn? IEND?
```

The regular operators $*$ and ? have the usual meaning of repetition and optionality. The identifiers (e.g., `Signature`, `IHDR`, etc.) refer to specific components of PNG. These structures are described in the second part of a DERRIC description. As an example, the following snippet describes the `IEND` structure:

```
IEND {
  length: 0 size 4;
  chunktype: "IEND";
  crc: 0xAE, 0x42, 0x60, 0x82
}
```

This declaration states that the `IEND` structure consists of a length field of 4 bytes (containing zeros), followed by the (ASCII encoded) string "IEND", and terminated by a CRC code consisting of 4 constant values. To factor out common fields in structure definitions, DERRIC allows structures to inherit from other structures. For instance, in PNG, most structures inherit from an abstract `Chunk` structure which declares common fields for length, type, data and CRC check; such fields can be overridden if needed.

A DERRIC description is input to the DERRIC compiler which generates executable *validators*. A validator tries to match binary input streams against the file format definition captured in DERRIC. One application of these validators is *file carving*: the process of recovering possibly damaged or fragmented files from storage

devices [Coh07, PM09]. Previous research has shown that the generated validators perform well, both in terms of recovered files and runtime speed (see Chapter 3, and that Derric descriptions can be automatically transformed to improve runtime performance (see Chapter 4.

The benefits of Derric are only fully realized, however, if the file format description can be considered correct. If files are encountered that are not recognized, there are two possibilities:

- The binary data is not an instance of the file format we are looking for, or the data is corrupted. In other words, the data is at fault.
- The file format description is incorrect and has to be changed to cope with this specific variation of the file format.

Note that these situations may overlap. In fact, it is quite common to relax a file format description to trade some precision for a higher recall. Nevertheless, in both cases the question remains: how to find out if a description should be adapted to the new situation? And if so, how should the description be changed? Trinity helps to answer such questions by providing debugger functionality at the level of Derric itself. This way, both the data and the runtime state of an analysis can be interpreted in terms of the sequential layout and the structures and fields of the file format.

6.2 Trinity

Integrated Data Debugging

Trinity is an IDE which aims to leverage the domain-specific information contained in Derric descriptions to bring integrated data debugging support to the process of reverse engineering binary file formats. A screen shot of Trinity is shown in Figure 6.1. The IDE consists of three synchronized views:

- **Data**: A hexview showing the input data (top right).
- **State**: An outline view of the runtime state, with root nodes for structures and child nodes for fields (left column).
- **Code**: A syntax-highlighting editor for showing a Derric description (bottom right).

The user can navigate between views using hyper links which connect all three views. For instance, after selecting the byte at offset 8 in the Data view at the top right, the contextual structure and field of this byte are highlighted. Similarly, the `IHDR` structure and its `length` field are highlighted in the State view on the left,

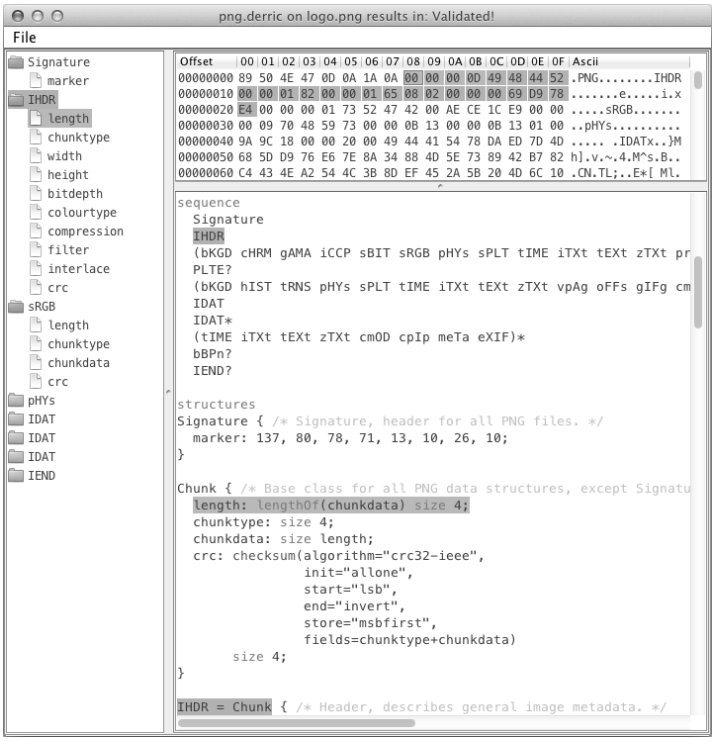

Figure 6.1: Screenshot of TRINITY used on a PNG example file.

which provides the dynamic execution context to this byte. In the Code view at bottom right, the `IHDR` structure is highlighted in both the **`sequence`** and **`structures`** sections. Finally, the `length` field is highlighted in the Code view as well, where it is defined not directly in the `IHDR` structure, but in the `Chunk` structure it inherits from.

It is also possible to go the other way. For instance, clicking on a field in the code view will highlight all the bytes in the input stream that have been successfully matched using that very field. Similarly, clicking on an element in the sequence section highlights all bytes in the input stream captured by that syntactic element. Because syntactic elements in the sequence may occur multiple times (through the use of the regular operator $*$), clicking on a source element may highlight multiple parts of the input data.

Figure 6.2 illustrates the relationships between the three views in more detail. On the left (Data) is a hexview of the input data (between offsets 16 (`0x0010`) and 48 (`0x002C + 4`). In the center (State) the trace of interpreting the input data (showing

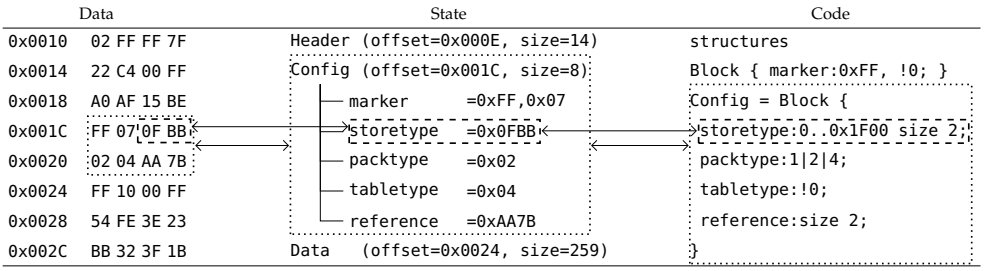

Figure 6.2: The relationship between Data view (left, hexview), State view (center, outline) and Code view (right, text editor).

matches for structures named `Header`, `Config` and `Data`, of which only `Config` is expanded and showing its fields). On the right (Code) the text editor view of the Derric description (showing the definition of the `Config` structure). In all three views, the dotted line marks the `Config` structure and the dashed line its `storetype` field.

By making the links between data, runtime state and code explicit, Trinity simplifies the reverse engineering and maintenance tasks in dealing with binary file formats. The developer can interactively explore the original file format description in Derric directly in the context of the actual bytes in the input data. Below we describe how Trinity can be used in digital forensics practice.

A File Format Reverse Engineering Scenario

The design of Trinity is informed by more than a decade of experience in reverse engineering file formats. Additionally, in previous research we have performed an experiment which studied corrective maintenance of Derric descriptions (see Chapter 5[1] by executing evolution scenarios. These scenarios for "fixing" the descriptions all represent typical cases where Trinity could be used. In fact, the research of Chapter 5 would have been much less time consuming if Trinity had been available at the time, as most of the effort consisted of relating error locations in binary data to source locations in Derric.

The use of Trinity starts when a file is encountered that is expected to validate, but fails to do so. The following steps describe the expected work flow using Trinity:

[1]The changes can be reviewed online at `http://github.com/jvdb/derric-eval/`.

Initial Run

The file and the Derric description of its expected file format are loaded into Trinity and the interpreter halts at the first byte where validation fails (i.e., where a value is encountered that does not match the description). The file's contents is shown in the Data view, the Derric description in the Code view and the generated trace after an initial run in the State view.

Locate Area of Interest

The user clicks on the last data structure listed in the trace, automatically showing the relevant child nodes. The Data view is automatically scrolled to the corresponding bytes. The cursor in the Code view is positioned on the structure where validation failed.

Inspect Structure

The user clicks the last field below the structure in the trace. This keeps the existing highlighting but adds additional ones of the fields' bytes in the Data view and its description in the Code view.

Make Corrections

Based on whether that field is the source of the validation error, the user will either make a modification or move up to the previous field, backtracking until a field or structure is encountered which accounts for the failure. Finally, the validation is rerun, and the process repeats if there are (new) failures.

6.3 Implementation

Derric is implemented as an external DSL in the metaprogramming language Rascal [KSV09a]. Rascal provides built-in grammars for describing syntax, primitives for analyzing and transforming source code, and provides hooks into the Eclipse IDE to obtain editor services (e.g., syntax coloring, outlining, hyperlinking etc.).

The Derric compiler operates in three steps. First the Derric description is desugared (e.g., flattening inheritance, constant propagation). Second, a Derric description is transformed to an intermediate representation called Validator, which is an imperative but platform-independent model of the final validator. Finally, the Validator model resulting from the previous step is transformed to Java source code.

An overview of the architecture of Trinity is shown in Figure 6.3. Trinity reuses the front-end part of the Derric compiler, up to and including the transformation

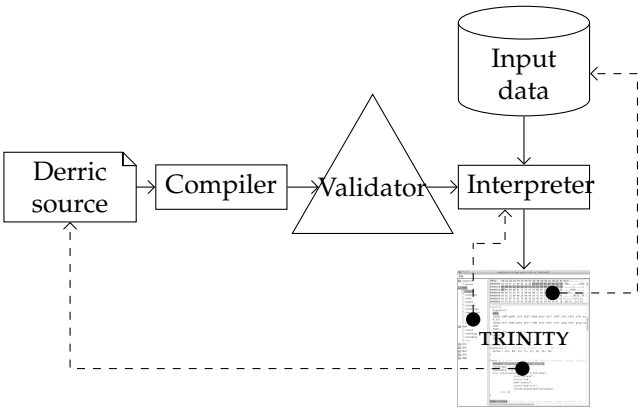

Figure 6.3: The Trinity architecture. The dashed arrows indicate the information sources of the three views in the IDE.

to the Validator model. Instead of generating Java code however, the Java foreign-function interface of Rascal is used to build an in-memory model in Java of the Validator. Following the Interpreter design pattern, the classes representing the model contain evaluation methods to execute the validator. This interpreter is then hooked up to the Trinity IDE.

To realize the fine-grained cross-linking of views in Trinity, origin tracking is used [DKT93]. This means that the original source locations of syntactic elements in a Derric description are maintained throughout all phases of the compiler and interpreter. The Derric parser generated by Rascal initially annotates the parse tree with such origins. During desugaring and the transformation to the Validator model, the origins are propagated. Finally, the in-memory model in Java is decorated so that, when the interpreter is stopped, the Trinity runtime environment knows where in the Derric description execution is taking place. The same technique is used to maintain a mapping from the runtime state (i.e., the values of the matched structures and fields), to the source code, and from the source code to the data.

6.4 Related work

The key idea of Trinity is to integrate the input data into the activity of debugging and to provide bidirectional cross-links among code, state and data. Moreover, the integration is domain-specific: Derric descriptions capture file formats at a level that can be understood by forensic investigators. In Trinity this understandability extends to the data and the runtime state of the validator. As a result, Trinity pro-

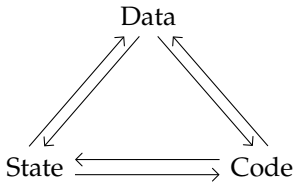

Figure 6.4: The trinity of debugging in Trinity.

vides debugging for reverse engineering file formats at a higher level of abstraction.

Using Trinity the user can navigate from the source code to the data and vice versa, but also from the runtime state to the data and vice versa, and finally, it is possible to go from the data to the runtime state and the source code. We have depicted these 6 types of cross links in Figure 6.4. Traditional debuggers, on the other hand, provide only two of such links: 1) from the runtime state (e.g., stack trace) to the source code, and 2) from the data to the code (e.g., from a variables view to declaration sites). Although specialized visualizations for general purpose debuggers are quite common (e.g., [ZL96, Hex]), these do not provide the same level of integration as Trinity.

Trinity is most related to domain-specific debuggers in other domains. For instance, ANTLRWorks [BP08] is an IDE which provides support for debugging ANTLR grammars. The generated parsers communicate with the IDE during their execution, allowing the user to replay its actions and inspect the input data, grammar and parse tree afterwards. Similar tools exist for debugging regular expressions. A recent example is Debuggex [Toa], which features coloring of the (matched) input data, and visualization of the finite-state automaton.

6.5 Conclusion and Future Work

Reverse engineering binary file formats is a time-consuming and error-prone activity. One of the reasons is that the relation between the structure of the data and how software processes that data is obscured by low-level implementation details and has to be mentally reconstructed. In this chapter we have presented Trinity, an IDE that brings integrated data debugging support to the Derric IDE for file format description. It consists of three views, which display the input data, the runtime state of a file format validator and the Derric source code respectively. Each view is related to the other. Clicking in any of the views highlights corresponding elements in the others. If a file fails to validate, the three integrated views allow the developer to assess the situation: why does validation fail? What changes are needed to the file format description? Trinity aims to reduce the effort of performing corrective

maintenance of digital forensics software.

There are ample opportunities for further improving Trinity. For instance, the way elements are highlighted in the different views is mostly syntactic. One extension would be to add more semantics to the visualization. For instance, clicking a field that has a dependency on another field in its length or content specification, could also highlight the bytes that were captured by those dependency fields. Conversely, clicking on a byte in the data view could also trigger highlighting of all expressions affected by it. Such data flow visualization could further increase understanding of what happens at runtime and help diagnosing failures.

Another direction for further work is increasing the "liveness" of Trinity [LF95]. Currently, Trinity allows the dynamic inspection of state and data. However, changes to the Derric description still requires a full rerun of the validator. The potential benefits presented by Trinity could be increased further by instantly reflecting a change to the format description in the other views. One way to approach this is to incrementally update the runtime state of the interpreter based on the changes to the code (see, e.g., [Sto13]).

Finally, we plan to perform a user study to evaluate to what extent Trinity helps to improve the maintenance of Derric descriptions. The evolution scenarios obtained in Chapter 5 can provide a starting point for the maintenance tasks to set up this experiment.

Part IV

Retrospective

Highlights:

- Summary of contributions.
- Overall conclusions of the research.

CHAPTER 7

Contributions

This thesis addresses the main research question posed in Chapter 1:

> *Can we improve the practice of engineering automated digital forensics tools through the application of model-driven software engineering techniques, specifically in the domain of recovering information stored in files?*

In order to specify the meaning of *improve* in the context of this question, we have broken it down into four specific research questions. In the following sections we discuss the research results that contribute to answering these questions.

7.1 Achieving Separation of Concerns

Q1: Can we separate the concerns in file format specification from their implementation?

We consider Derric a sufficient example of the feasibility of separating the specification of a file format from its implementation in the domain of automated digital forensics. Although our evaluations discuss only a small set of file formats described in Derric, the used file formats are diverse in structure and representative of file formats in general.

Domain Analysis

In Chapter 1 we described that the highest level of variability in data storage exists at the level of application file formats. New file formats are encountered regularly,

as well as multiple versions and variants of existing file formats. In order to simplify development and maintenance of the recovery tools that validate files in those file formats, we have built the DSL Derric.

We distinguish between requirements for the features of the DSL (i.e., what should be expressible in the DSL) and requirements for the syntax of the DSL (i.e., what the surface syntax should look like). In order to design Derric, we have performed a domain analysis to determine the requirements for both aspects.

Language Features

In Chapter 2 we discussed the aspects of file formats that are relevant for the engineering of recovery tools. This domain analysis creates a constraint on the design of the resulting DSL: any of those aspects should be expressible in some form, in order to guarantee that the associated recovery tool can be created with it. As a result, we arrive at the following three requirements:

Constant specification: To allow *Magic Number Matching*, the DSL should support the specification of constant values occurring in fixed locations in a file format.

Data dependencies: To allow *Data Dependency Resolving*, the DSL should support the specification of dependencies between the values of fields.

Content analysis: To allow *Internal Verification Checking*, *Output Analysis* and *Data Decoding*, the DSL should support the specification of such algorithms.

A design decision we have made is to design Derric with direct support for constant specification and data dependencies, but without direct support for content analysis. Any type of content analysis can be specified in Derric, but only by naming it and providing configuration values (such as encoding) and data dependencies (such as compression tables).

There are two reasons for this decision. First, this design allows Derric to be a small and fully declarative language, that is easy to automatically analyze and transform. Second, this type of specification allows content analysis to be easily mapped to an existing implementation, reducing the amount of work related to specifying a file format.

This decision comes with a major drawback: the implementation of output analysis cannot easily be included in automated analyses and transformations, since they are not expressed in Derric. Realizing specification of content analysis algorithms in Derric is future work, since it may allow higher fidelity optimizations.

For example, with support for specifying these algorithms, it is conceivable that the transformations discussed in Chapter 4 would have lead to higher granularity control of the resulting tool performance.

Language Syntax

Apart from the required features for specifying values, the actual syntax of Derric is also based on a domain analysis, discussed in Chapter 3. The notation is based on the most common sources of information that lead to the development and maintenance of Derric specifications: reverse engineering, documentation and source code.

Reverse engineering usually leads to the use of data dump utilities showing data in hexadecimal encoding. Documentation often employs pseudocode, and source code to file format specifications is usually in C/C++ and Java. Staying close to that familiar syntax is beneficial, since these languages are most used for low-level serialization code that writes data in binary file formats.

7.2 Measuring Runtime Performance Costs

Q2: Can we determine what the runtime performance costs are of separating the concerns of file format specification from their implementation?

In our implementation and evaluation, we have not observed any runtime performance penalties resulting from separating the concerns of file format specification and implementation.

Evaluating Runtime Performance

Discussed in Chapter 3, we have created the file carver Excavator. It implements similar functionality as a set of three popular file carvers. Excavator uses code generated by the Derric compiler for its file format validation concern.

The tools used in our comparison were selected based on two criteria. First, all tools should be used in practice, to ensure relevancy of the comparison. Second, all tools should be open source, in order to compare the implementation to that of Excavator/Derric. The second criterium did not lead to the exclusion of any relevant tools.

The comparison was performed on a set of five benchmarks. Three of the benchmarks are part of the Digital Forensics Tool Testing-suite (DFTT) and the other two were Forensic Challenges at the Digital Forensics Research Workshops (DFRWS) in 2006 and 2007.

The benchmarks used in our comparison were selected based on three criteria. First, the benchmarks should be publicly available to ensure the possibility of reproduction. Second, we chose to use only existing benchmarks because they were previously used in other comparisons and are not biased towards our evaluation. Finally, the benchmarks all contain files in the JPEG file format, since that is the file format specification that we had developed fully.

Only one of the competing tools (ReviveIt) was capable of recovering a small amount of additional files on one of the benchmarks. In the area of runtime performance Excavator's performance was comparable to the others. Based on this, we conclude that Excavator/Derric performs similarly to existing tools and that its MDSE-approach does not impact its performance negatively.

Additionally, Chapter 3 contains a qualitative assessment of the implementations of the competing tools and how they compare to that of Excavator/Derric. We found that all existing tools that implement at least one reconstruction algorithm (ReviveIt and PhotoRec) tangle the concerns of validating file formats and reconstruction algorithms, the different concerns of a file carver as discussed in Chapter 2. The high amount of variability in file format specifications will lead to much higher maintenance costs in those tools than in Excavator/Derric.

7.3 Leveraging Model Transformation

Q3: Can we leverage model transformation to tune the scalability and runtime performance of our solution?

The model transformations we have implemented lead to the fully automated generation of several implementations with different runtime performance characteristics. This allows the user to make the trade-off between precision and runtime performance.

Custom Benchmark Development

Since the largest file carving benchmark available at the time of our evaluation in Chapter 3 was 331MB, we concluded that it was not possible to evaluate the scalability of Excavator/Derric.

In order to study scalability, we have constructed a 1TB benchmark. It contains 357GB of image files of the types JPEG, PNG and GIF, downloaded from Wikipedia using the latest static file listing from 2008. These files were spread out across the 1TB image along with blocks of zero and random data. Additionally, a small percentage of the files were fragmented so that they would correspond to the fragmentation levels observed in an empirical study of hard drive fragmentation in practice [Gar07].

Since Wikipedia contains data from a large amount of different sources, the resulting benchmark is not biased towards some specific version or variant of the file formats used. We have verified this diversity by investigating the metadata of the image files, as discussed in Chapter 5. 28.4% of all image files contained an EXIF *Software* metadata tag, specifying a total of 4,024 different origins (including platforms, applications, versions and variants).

Configurable Performance Trade-Offs

To allow users fine-grained control over the runtime performance characteristics of Excavator/Derric when investigating large amounts of data, we have implemented three model transformations. These transformations automatically remove and modify Derric descriptions, to make the resulting validation less strict.

We hypothesized that this reduced strictness would lead to increased runtime performance in exchange for reduced precision. Making this trade-off configurable allows investigators to iterate between data recovery and information analysis, as described in Chapter 1. The transformations were implemented in order to be successively executed: first removing content analysis (NoCA), then removing data dependency resolving (NoDD) and finally reducing the description to a constant header and footer definition (Header).

The transformations were intended to create a kind of dial to turn in order to control the runtime performance of Excavator/Derric. However, we observed that all three levels actually lead to comparable precision and runtime performance levels. Still, the transformed descriptions lead to considerable increased runtime performance (realizing a speed up between 40% and 320%) at the cost of only 8% loss in precision and 5% loss in recall.

As a result, the intended dial ended up as a switch. However, it is fully automatic and leads to two versions of the same tool, that are both useful in practice and allows users to make decisions about how to perform an investigation without having to manually perform modifications to the tool first.

Our evaluation only considers the transformations in a fixed order: first running NoCA, then NoDD and finally Header. Future work may explore the results of performing these transformations without each other. Additionally, extensions to Derric to allow the expression of content analysis algorithms may lead to potential transformations that modify parts of the algorithms in such ways to still allow a dial-like control over the resulting tools.

7.4 Evaluating Maintainability

Q4: Can we determine whether our solution provides the modifiability required in practice?

Our set of Derric descriptions could be modified to support all variants of a large corpus of image files without requiring modifications to the language itself. Some complicated and cross-cutting changes were required however, which lead to the identification of a set of language features to improve Derric.

Realistic Maintenance Scenarios

Discussed in Chapter 5, we discovered that our Derric descriptions of JPEG, PNG and GIF did not cover the entire set of files in our Wikipedia corpus used in Chapter 4. In order to evaluate whether Derric would be usable in practical maintenance scenarios, we isolated all files that did not match our descriptions. This resulted in a set of 11,663 files, corresponding to 1.0% of the entire corpus.

We hypothesized that if Derric is a suitable DSL for this the domain, that we should be able to correct all errors without significant changes to the DSL implementation. Furthermore, that the changes would be easily expressible as localized modifications to the descriptions.

Repairing all descriptions required a total of 37 changes: 11 for JPEG, 8 for GIF and 18 for PNG. It was successful: all 11,663 files, as well as the rest of the corpus, were recognized correctly after these changes were applied. Additionally, all required changes were confined to the Derric descriptions, so no changes were made to the implementation of Derric itself.

Evidence-Based DSL Evolution

Discussed in Chapter 5, we classified all required changes into three categories based on their *change complexity*, distinguishing between low, medium and high complexity. Low complexity means single, localized edits to the descriptions and consisted of 13 of the required changes. These changes are ideal, since they cleanly map a single required change to a single change in the descriptions.

Medium complexity refers to multiple, but dependent edits and consisted of 19 of the required changes. While having to perform multiple changes to express a single change is not preferred, it can be the result of a conscious design decision in the DSL, as is the case here. In Derric, ordering of data structures is separated from description of these data structures. The result is that adding a new data structure to a description requires at least two modifications: one to specify the data structure and one to add it to the sequence definition.

Finally, five high complexity changes remained. They all required multiple, cross-cutting changes to be performed on the JPEG description. In order to prevent these changes from amplifying the complexity of future changes to the descriptions, we propose to add three language features. Adding these features to Derric eliminates the need for the high complexity changes observed.

To reduce duplication in the description's sequence definition, we propose to add an abstraction mechanism to factor out common subsequences. In a similar vein, to prevent duplication in order to express precedence rules, we propose to also add a precedence operator to the sequence definition. The final proposed feature is highly domain-specific: padding. In binary file formats, padding is a

common construct, so it makes sense to add this as a native language feature to Derric.

Advanced Tool Support

Discussed in Chapter 6, we have developed Trinity, an integrated development environment (IDE) aimed at debugging Derric descriptions. Both our analyses of the domain discussed in Chapters 2 and 3, as well as our experience in performing maintenance discussed in Chapter 5 stress the importance of inspecting data dumps when describing file formats.

In regular implementations of file formats, such as in general-purpose programming languages, the relationship between individual locations in a data dump and the code implementing the related data structure can be difficult to determine.

We have resolved this in Trinity through the use of origin tracking. Locations on both the sequence and data structures are propagated through all stages of compilation and interpretation. Additionally, we also annotate the input stream with the location information of the code that is interpreting it, in order to create a mapping between input data, interpreter state and Derric source locations.

The result is an IDE that provides a continually synchronized view of all related inputs: clicking anywhere in the data highlights the associated items in the state and Derric source. The synchronization works in the other directions as well: starting from the state or source code will highlight all associated items in the other views.

When performing maintenance, a Derric user can simply load a file that does not validate into Trinity and inspect either the location in the input data or the data structure in Derric that causes an error. Clicking on any of these will automatically show all related locations, reducing the manual tracking or debugging otherwise required.

Languages such as C/C++ and Java rely on imperative code to map input values to data structures. Realizing similar functionality in an IDE for those languages is not possible due to the undecidable nature of the static analysis required.

In contrast, Rascal provides a simple mechanism for managing and propagating location information, that Trinity uses extensively. It does require the developer of the Derric compiler and interpreter to maintain the mapping for each transformation, but this is essential complexity.

CHAPTER 8

Conclusions

The output of the research in this thesis is threefold. First, it is an extensive case study in model-driven software engineering. Second, it is a case study in applying model-driven software engineering in the domain of automated digital forensics, through the design and use of DERRIC and associated tools. Finally, it is also a case study in applying RASCAL in the domain of DSL engineering, as RASCAL was used to prototype, implement and evolve many different parts of DERRIC and TRINITY.

In this chapter we discuss the lessons learned from each of these three perspectives and discuss future research directions for continuing work in this area.

8.1 Model-Driven Software Engineering in Practice

We consider the research described in this thesis as an extensive case study in model-driven software engineering. It adds to the growing body of knowledge of practical experience with MDSE as discussed in Chapter 1 and contributes results in some specific areas.

Our results include clear indications of increased maintainability and greater flexibility, resulting in reduced development times. At the same time, whether this leads to lower total costs is an open question, since this will depend on how the maintenance costs associated with maintaining the DERRIC compiler and tools compare to the higher productivity and maintainability of using them.

Additionally, using DERRIC enables the construction of custom tools and the use of model transformations to implement automated domain-specific optimizations. These fall clearly within the area of AVOPT [MHS05], showing the relevance of this decision pattern in practice.

8.2 Derric: Applying MDSE in Automated Digital Forensics

Applying MDSE in the domain of automated digital forensics can be considered a clear success, based on the results presented in this thesis. In Chapter 1 we discussed how the software engineering challenges in automated digital forensics revolve around the non-functional requirements of runtime performance, scalability and modifiability.

We have shown in direct comparison to existing tools on standard benchmarks that our approach increases the modifiability of the solution, while not losing in terms of runtime performance. Furthermore, we have shown how Derric can be used to increase scalability and runtime performance through automated domain-specific optimizations. Additionally, it allows the construction of domain-specific tools that would be prohibitively complex to develop otherwise.

Finally, its modifiability is not only shown as a result of its clear technical separation of concerns. We have also demonstrated its practical use in realistic maintenance scenarios, which resulted in small improvements to the language.

At the same time, the domain of automated digital forensics is large and diverse, so we have confined our evaluation to the subdomain of file carving. Many more tools in automated digital forensics use file validators, so we expect that a lot of our work can quickly be carried over for use in those subdomains.

Furthermore, while many file formats exist that are relevant in forensic investigations, in our evaluations we have only considered the most common image file formats JPEG, PNG and GIF. Although we have described many other file formats in Derric, they were excluded from our current evaluations for practical reasons.

8.3 Rascal: DSL Engineering in Practice

Rascal is itself a DSL, in the domain of software analysis and transformation, which includes DSL engineering [KSV09a]. Although it has been successfully used to analyze and transform software [HKSV11b, HKV13], the research in this thesis describes the first large application in DSL engineering.

Based on the experiences with Derric, we can draw two conclusions about the design goals of Rascal [BHK+11]. First, as a meta-programming language it succeeded in allowing the development of Derric, without relying on any external tools or libraries for any of its tasks.

This included syntax definition, parsing, transformation, code generation and interpretation. The actual interpreter is implemented in Java both for reasons of performance and interoperability. The Derric compiler consists of 2,346 lines of Rascal code, which can be considered highly compact given its functionality.

Second, Rascal's goal to be a relatively simple language that allows its users to slowly discover and use its more powerful features is clearly visible in the history

of the Derric compiler's code base[1]. Initial versions of the compiler resemble Java programs while the final version contains several parts using some of Rascal's more powerful features.

While the Trinity IDE was implemented in Java, a prototype was first developed and debugged in Rascal using its visualization library.

8.4 Future Directions

Two directions for future research are the direct result of this thesis. The first as the continuation of this work, further improving and evaluating Derric in different settings. The second as a result of the achievements in this work, to attempt to explore new opportunities created by the development of Derric.

Derric in Practice

Further evaluating the suitability and applicability of Derric can be realized through the construction of a larger and more diverse corpora of inputs, describing more file formats in Derric, and constructing additional tools in related subdomains. However, since our research is the result of a practical need instead of an overarching research program, our intentions are to evaluate Derric further through deployment and application in the daily practice of automated digital forensics.

This will eventually result in similar evaluations, since practical application will confront the tools using Derric with a large amount of inputs, require additional file formats to be described and perfective maintenance to be performed. A major advantage however is that these evaluations will be driven strictly by practical needs, ensuring applicability of the results.

We especially expect input on the general usability of both Derric and Trinity through the use of them by forensic investigators and other domain experts in practice. Additionally, we expect results on the validity of our approach for applying transformations (see Chapter 4).

Exploring Opportunities

Besides improving the engineering of automated digital forensics tools, Derric has the potential to fundamentally improve the capabilities of tools in this domain. While we have not explored these opportunities in this thesis, there are several areas that may yield significant results.

The declarative nature of Derric descriptions opens the possibility for many analysis scenarios. For example, instead of generating file validators that are uti-

[1] Available from `http://www.cwi.nl/model-driven-engineering-in-digital-forensics`.

lized by reconstruction algorithms, hybrid approaches can be considered where each file format may result in a custom reconstruction approach.

Furthermore, while manually maintaining a single validator that recognizes multiple file formats is difficult, having declarative descriptions separately from their implementation opens up the possibility of using alternate approaches to achieve similar results. First, a generative approach that merges Derric descriptions may have considerable advantages such as lower memory requirements. Second, a stream-oriented approach may exploit opportunities for extensive parallelization. To enable the exploration of such enhancements, initial future work will focus on formalizing the semantics of Derric.

Bibliography

[AB11] Leon Aronson and Jeroen van den Bos. Towards an Engineering Approach to File Carver Construction. In *2011 IEEE 35th Annual Computer Software and Applications Conference Workshops (COMPSACW)*, pages 368–373. IEEE, 2011. (page 18, 19)

[AC72] Frances Allen and John Cocke. A Catalogue of Optimizing Transformations. In *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1972. (page 72)

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Prentice Hall, 2 edition, 2006. (page 67)

[Alv11] Lizette Alvarez. Software Designer Reports Error in Anthony Trial. *The New York Times*, July 2011. `http://www.nytimes.com/2011/07/19/us/19casey.html`. (page 3)

[AT05] Ahmad Almulhem and Issa Traore. Experience with Engineering a Network Forensics System. In Cheeha Kim, editor, *Proceedings of the International Conference on Information Networking, Convergence in Broadband and Mobile Networking (ICOIN'05)*, volume 3391 of *Lecture Notes in Computer Science*, pages 62–71. Springer, 2005. (page 14)

[Axe10] Stefan Axelsson. The Normalised Compression Distance as a File Fragment Classifier. *Digital Investigation*, 7(S1):24–31, 2010. Proceedings of the Tenth Annual DFRWS Conference. (page 24)

[Bac02] Godmar Back. DataScript—A Specification and Scripting Language for Binary Data. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 66–77. Springer, 2002. (page 54)

[BBB+12] Raoul A. F. Bhoedjang, Alex R. van Ballegooij, Harm M. A. van Beek, John C. van Schie, Feike W. Dillema, Ruud B. van Baar, Floris A. Ouwendijk, and Micha Streppel. Engineering an Online Computer Forensic Service. *Digital Investigation*, 9(2):96–108, 2012. (page 14, 15)

[BBI+04] Grady Booch, Alan W. Brown, Sridhar Iyengar, James Rumbaugh, and Bran Selic. An MDA Manifesto. *Business Process Trends/MDA Journal*, May 2004. (page 10)

[BCK12] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, third edition, 2012. (page 4)

[Bee09] Nicole Beebe. Digital Forensic Research: The Good, the Bad and the Unaddressed. In Gilbert L. Peterson and Sujeet Shenoi, editors, *Revised Selected Papers from Advances in Digital Forensics V - Fifth IFIP WG 11.9 International Conference on Digital Forensics*, volume 306 of *IFIP Advances in Information and Communication Technology*, pages 17–36. Springer, 2009. (page 13)

[Béz06] Jean Bézivin. Model Driven Engineering: An Emerging Technical Space. In *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 36–64. Springer, 2006. (page 10, 72)

[BHK+11] Jeroen van den Bos, Mark Hills, Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. Rascal: From Algebraic Specification to Meta-Programming. In Francisco Durán and Vlad Rusu, editors, *Proceedings Second International Workshop on Algebraic Methods in Model-based Software Engineering (AMMSE'11)*, volume 56 of *Electronic Proceedings in Theoretical Computer Science*, pages 15–32, 2011. (page 114)

[BJMH02] Don Batory, Clay Johnson, Bob MacDonald, and Dale von Heeder. Achieving Extensibility through Product-Lines and Domain-Specific Languages: A Case Study. *ACM Transactions on Software Engineering and Methodology*, 11(2), April 2002. (page 11)

[BJS10] Marius Bozga, Mohamad Jaber, and Joseph Sifakis. Source-to-Source Architecture Transformation for Performance Optimization in BIP. *IEEE Transactions on Industrial Informatics*, 6(4):708–718, 2010. (page 72)

[BK05] Jeroen van den Bos and Ronald van der Knijff. TULP2G: An Open Source Forensic Software Framework for Acquiring and Decoding Data Stored in Electronic Devices. *International Journal of Digital Evidence*, 4(2), 2005. (page 14)

[BLW05] Paul Baker, Shiou Loh, and Frank Weil. Model-Driven Engineering in a Large Industrial Context—Motorola Case Study. In *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MODELS'05)*, volume 3713 of *Lecture Notes in Computer Science*, pages 476–491. Springer, 2005. (page 54)

[BP08] Jean Bovet and Terence Parr. ANTLRWorks: an ANTLR grammar development environment. *Software: Practice & Experience*, 38(12):1305–1332, 2008. (page 101)

[BRLM07] Laurent Burgy, Laurent Reveillere, Julia L. Lawall, and Gilles Muller. A Language-Based Approach for Improving the Robustness of Network Application Protocol Implementations. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems (SRDS'07)*, pages 149–160, 2007. (page 54)

[BS11] Jeroen van den Bos and Tijs van der Storm. Bringing Domain-Specific Languages to Digital Forensics. In *33rd International Conference on Software Engineering (ICSE'11)*, pages 671–680. ACM, 2011. (page 18, 35)

[BS12] Jeroen van den Bos and Tijs van der Storm. Domain-Specific Optimization in Digital Forensics. In Zhenjiang Hu and Juan de Lara, editors, *5th International Conference on Model Transformation (ICMT'12)*, volume 7307 of *Lecture Notes in Computer Science*, pages 121–136. Springer, 2012. (page 18, 57)

[BS13a] Jeroen van den Bos and Tijs van der Storm. A Case Study in Evidence-Based DSL Evolution. In Pieter Van Gorp, Tom Ritter, and Louis M. Rose, editors, *9th European Conference on Modelling Foundations and Applications (ECMFA'13)*, volume 7949 of *Lecture Notes in Computer Science*, pages 207–219. Springer, 2013. (page 18, 77)

[BS13b] Jeroen van den Bos and Tijs van der Storm. TRINITY: An IDE for The Matrix. In *29th IEEE International Conference on Software Maintenance (ICSM'13)*, pages 520–523. IEEE, 2013. (page 18, 93)

[BV04] Martin Bravenboer and Eelco Visser. Concrete Syntax for Objects: Domain-Specific Language Embedding and Assimilation without Restrictions. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383. ACM, 2004. (page 12)

[Car] Brian Carrier. Digital Forensics Tool Testing Images. `http://dftt.sourceforge.net/`. (page 48)

[Car05] Brian Carrier. *File System Forensic Analysis*. Addison-Wesley, 2005. (page 5, 7)

[Cas09] Eoghan Casey, editor. *Handbook of Digital Forensics and Investigation*. Academic Press, 2009. (page 5)

[CBDM01] Eui-Young Chung, Luca Benini, and Giovanni De Micheli. Source Code Transformation based on Software Cost Analysis. In *Proceedings of the 14th International Symposium on Systems Synthesis (ISSS'01)*, pages 153–158. ACM, 2001. (page 72)

[CBS+10] Gregory Conti, Sergey Bratus, Anna Shubina, Benjamin Sangster, Roy Ragsdale, Matthew Supan, Andrew Lichtenberg, and Robert Perez-Alemany. Automated Mapping of Large Binary Objects using Primitive Fragment Type Classification. *Digital Investigation*, 7(S1):3–12, 2010. Proceedings of the Tenth Annual DFRWS Conference. (page 24)

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, 2000. (page 72)

[Cen09] Centraal Bureau voor de Statistiek. *De Digitale Economie*. 2009. In Dutch. (page 40)

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–646, 2006. (page 47)

[Coh07] Michael I. Cohen. Advanced Carving Techniques. *Digital Investigation*, 4(3-4):119–128, 2007. (page 23, 45, 63, 73, 96)

[DK98] Arie van Deursen and Paul Klint. Little Languages: Little Maintenance? *Journal of Software Maintenance*, 10(2):75–92, 1998. (page 11, 78)

[DKT93] Arie van Deursen, Paul Klint, and Frank Tip. Origin tracking. *Journal of Symbolic Computation*, 15:523–545, 1993. (page 100)

[DKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000. (page 11, 54, 78)

[DRIP12] Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Coupled Evolution in Model-Driven Engineering. *IEEE Software*, 29(6):78–84, 2012. (page 91)

[ERKO11] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. SugarJ: Library-based Syntactic Language Extensibility. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'11)*, pages 391–406. ACM, 2011. (page 12)

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring*. Addison-Wesley, 1999. (page 84)

[FGLP10] Jean-Marie Favre, Dragan Gasevic, Ralf Lämmel, and Ekaterina Pek. Empirical Language Analysis in Software Linguistics. In Brian A. Malloy, Steffen Staab, and Mark van den Brand, editors, *Third International Conference on Software Language Engineering (SLE'10)*, volume 6563 of *Lecture Notes in Computer Science*, pages 316–326. Springer, 2010. (page 90)

[FMW10] Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. The Next 700 Data Description Languages. *Journal of the ACM*, 57(2):1–51, 2010. (page 41, 54)

[Fow10] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley, 2010. (page 12)

[Gar07] Simson L. Garfinkel. Carving Contiguous and Fragmented Files with Fast Object Validation. *Digital Investigation*, 4(S1):2–12, 2007. Proceedings of the Seventh Annual DFRWS Conference. (page 8, 20, 23, 25, 29, 44, 45, 48, 60, 61, 69, 73, 108)

[Gar10] Simson L. Garfinkel. Digital Forensics Research: The Next 10 Years. *Digital Investigation*, 7(S1):S64 – S73, 2010. Proceedings of the Tenth Annual DFRWS Conference. (page 5, 13, 37, 73)

[Gar13] Simson L. Garfinkel. Digital Media Triage with Bulk Data Analysis and bulk_extractor. *Computers & Security*, 32(0):56–72, 2013. (page 14)

[GJ08] Dick Grune and Ceriel Jacobs. *Parsing Techniques—A Practical Guide*. Springer, 2008. (page 55)

[GKP07] Boris Gruschko, Dimitris S. Kolovos, and Richard F. Paige. Towards Synchronizing Models with Evolving Metamodels. In *Proceedings of the 2007 International Workshop on Model-Driven Software Evolution*, 2007. (page 91)

[Gre09] Christophe Grenier. PhotoRec, 2009. `http://www.cgsecurity.org/wiki/PhotoRec`. (page 47, 73)

[HBJ09] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Jürgens. COPE - Automating Coupled Evolution of Metamodels and Models. In Sophia Drossopoulou, editor, *23rd European Conference on Object-Oriented Programming (ECOOP'09)*, volume 5653 of *Lecture Notes in Computer Science*, pages 52–76. Springer, 2009. (page 91)

[Hex] Hex Rays. IDA. `https://www.hex-rays.com/products/ida/index.shtml`. (page 101)

[HKSV11a] Mark Hills, Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. A Case of Visitor versus Interpreter Pattern. In Judith Bishop and Antonio Vallecillo, editors, *49th International Conference on Objects, Models, Components and Patterns (TOOLS'11)*, volume 6705 of *Lecture Notes in Computer Science*, pages 228–243. Springer, 2011. (page 90)

[HKSV11b] Mark Hills, Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. A Case of Visitor versus Interpreter Pattern. In Judith Bishop and Antonio Vallecillo, editors, *49th International Conference on Objects, Models, Components, Patterns (TOOLS'11)*, volume 6705 of *Lecture Notes in Computer Science*, pages 228–243. Springer, 2011. (page 114)

[HKV13] Mark Hills, Paul Klint, and Jurgen J. Vinju. An Empirical Study of PHP Feature Usage: A Static Analysis Perspective. In Mauro Pezzè and Mark Harman, editors, *International Symposium on Software Testing and Analysis (ISSTA'13)*, pages 325–335. ACM, 2013. (page 114)

[HPD09] Felienne Hermans, Martin Pinzger, and Arie van Deursen. Domain-Specific Languages in Practice: A User Study on the Success Factors. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, volume 5795 of *Lecture Notes in Computer Science*, pages 423–437. Springer, 2009. (page 11, 54, 90)

[ISO06] ISO/IEC 14764: Software Engineering–Software Life Cycle Processes–Maintenance, 2006. (page 80)

[ISO10] ISO/IEC/IEEE 24765:2010: Systems and Software Engineering – Vocabulary, 2010. (page 5)

[ITU92] ITU/CCITT. Recommendation T.81 (JPEG Compression Specification), 1992. (page 41)

[JMW10] Trevor Jim, Yitzhak Mandelbaum, and David Walker. Semantics and Algorithms for Data-Dependent Grammars. In *Proceedings of the 37th*

annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10), pages 417–430. ACM, 2010. (page 55)

[Kom09] Matt Komorowski. A History of Storage Cost, 2009. `http://www.mkomo.com/cost-per-gigabyte`. (page 40)

[KPP06] Dimitris S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Model Comparison: A Foundation for Model Composition and Model Transformation Testing. In *Proceedings of the 2006 International Workshop on Global Integrated Model Management (GaMMa'06)*, pages 13–20. ACM, 2006. (page 11)

[Kru92] Charles W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992. (page 11)

[KS06] Martin Karresand and Nahid Shahmehri. File Type Identification of Data Fragments by Their Binary Structure. In *IEEE Information Assurance Workshop*, pages 140–147, 2006. (page 20)

[KSV09a] Paul Klint, Tijs van der Storm, and Jurgen Vinju. Rascal: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of the Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'09)*, pages 168–177. IEEE Computer Society, 2009. (page 46, 62, 90, 99, 114)

[KSV09b] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. EASY Meta-programming with Rascal. In João M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva, editors, *International Summer School on Generative and Transformational Techniques in Software Engineering III (GTTSE'09)*, volume 6491 of *Lecture Notes in Computer Science*, pages 222–289. Springer, 2009. (page 6)

[KSV10] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. On the Impact of DSL Tools on the Maintainability of Language Implementations. In *10th Workshop on Language Descriptions, Tools and Applications (LDTA'10)*. ACM, 2010. (page 11, 90)

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Press, March 2008. (page 54)

[LF95] Henry Lieberman and Christopher Fry. Bridging the Gulf between Code and Behavior in Programming. In *Conference Proceedings of Human Factors in Computing Systems (CHI'95)*, pages 480–486. ACM, 1995. (page 102)

[LNTG05] Karl M. J. Lofgren, Robert D. Norman, Gregory B. Thelin, and Anil Gupta. Wear leveling techniques for flash EEPROM systems, May 2005. US Patent 6850443 B2. (page 13)

[LP10] Ralf Lämmel and Ekaterina Pek. Vivisection of a Non-Executable, Domain-Specific Language – Understanding (the Usage of) the P3P Language. In *IEEE 18th International Conference on Program Comprehension (ICPC'10)*, pages 104–113. IEEE, 2010. (page 90)

[LSVW10] Vincent Lussenburg, Tijs van der Storm, Jurgen J. Vinju, and Jos Warmer. Mod4J: A Qualitative Case Study of Model-Driven Software Development. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *13th International Conference on Model Driven Engineering Languages and Systems (MODELS'10) Part II*, volume 6395 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 2010. (page 88)

[Mar10] Shane Markstrum. Staking Claims: A History of Programming Language Design Claims and Evidence: A Positional Work in Progress. In *2nd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU'10)*, pages 7:1–7:5. ACM, 2010. (page 90)

[MC00] Peter J. McCann and Satish Chandra. Packet Types: Abstract Specification of Network Protocol Messages. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'00)*, pages 321–333. ACM, 2000. (page 54)

[ME05] Tom Mens and Amnon H. Eden. On the Evolution Complexity of Design Patterns. *Electronic Notes in Theoretical Computer Science*, 127(3):147–163, 2005. (page 90)

[Met] Joachim Metz. ReviveIt 2007. `http://sourceforge.net/projects/revit/`. (page 47)

[MFW+07] Yitzhak Mandelbaum, Kathleen Fisher, David Walker, Mary Fernandez, and Artem Gleyzer. PADS/ML: A Functional Data Description Language. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 77–83. ACM, 2007. (page 54)

[MH03] Mason McDaniel and M. Hossain Heydari. Content Based File Type Detection Algorithms. In *Hawaii International Conference on System Sciences*, page 332a, 2003. (page 24)

[MHOV12] Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. Evaluating the Design of the R Language - Objects and Functions for Data Analysis. In James Noble, editor, *26th European Conference on Object-Oriented Programming (ECOOP'12)*, volume 7313 of *Lecture Notes in Computer Science*, pages 104–131. Springer, 2012. (page 90)

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005. (page 11, 54, 72, 78, 90, 113)

[Mic08] Microsoft. Microsoft Office File Formats, 2008. `http://msdn.microsoft.com/en-us/library/cc313118.aspx`. (page 39)

[MN00] Mehryar Mohri and Mark-Jan Nederhof. Regular approximation of context-free grammars through transformation. In *Robustness in Language and Speech Technology*, chapter 9, pages 251–261. Kluwer, 2000. (page 73)

[Nat03] National Institute of Standards and Technology. *Software Write Block Tool Specification & Test Plan 3.0*, September 2003. (page 7)

[Nat04] National Institute of Standards and Technology. *Hardware Write Blocker Device (HWB) Specification 2.0*, May 2004. (page 7)

[Net05] Netherlands Forensic Institute. Defraser, 2005. `http://sourceforge.net/projects/defraser/`. (page 15)

[Oeh05] Peter Oehlert. Violating Assumptions with Fuzzing. *IEEE Security and Privacy*, 3(2):58–62, 2005. (page 80)

[Pal01] Gary Palmer. A Framework for Digital Forensic Science. In *A Roadmap for Digital Forensic Research*, pages 15–20, 2001. (page 5)

[Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007. (page 55)

[Pet95] Marian Petre. Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Communications of the ACM*, 38(6):33–44, June 1995. (page 11)

[PM09] Anindrabatha Pal and Nasir Memon. The Evolution of File Carving. *Signal Processing Magazine, IEEE*, 26(2):59–71, 2009. (page 4, 7, 20, 44, 58, 96)

[PV12] Richard F. Paige and Dániel Varró. Lessons Learned from Building Model-Driven Development Tools. *Software and System Modeling*, 11(4):527–539, 2012. (page 11)

[Ray91] Darrell R. Raymond. Characterizing Visual Languages. In *Proceedings of the 1991 IEEE Workshop on Visual Languages*, pages 176–182. IEEE, 1991. (page 11)

[RKPP10] Louis M. Rose, Dimitris S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Model Migration with Epsilon Flock. In Laurence Tratt and Martin Gogolla, editors, *3rd International Conference on Model Transformation (ICMT'10)*, volume 6142 of *Lecture Notes in Computer Science*, pages 184–198. Springer, 2010. (page 91)

[RR05] Golden G. Richard, III and Vassil Roussev. Scalpel: A Frugal, High Performance File Carver. In *Refereed Proceedings of the 5th Annual Digital Forensic Research Workshop (DFRWS'05)*, 2005. (page 20, 21, 47, 67, 73)

[RR06] Golden G. Richard, III and Vassil Roussev. Next-Generation Digital Forensics. *Communications of the ACM*, 49(2):76–80, 2006. (page 8)

[Sch06] Douglas C. Schmidt. Model-Driven Engineering. *Computer*, 39:25–31, 2006. (page 10, 54, 72)

[SM09] Husrev T. Sencar and Nasir Memon. Identification and recovery of JPEG files with missing fragments. *Digital Investigation*, 6(S1):88–98, 2009. Proceedings of the Ninth Annual DFRWS Conference. (page 22)

[Spi01] Diomidis Spinellis. Notable Design Patterns for Domain-Specific Languages. *Journal of Systems and Software*, 56(1):91–99, 2001. (page 11, 54)

[Sta06] Miroslaw Staron. Adopting Model Driven Software Development in Industry—A Case Study at Two Companies. In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MODELS'06)*, volume 4199 of *Lecture Notes in Computer Science*, pages 57–72. Springer, 2006. (page 54)

[Sto13] Tijs van der Storm. Semantic deltas for live DSL environments. In *First International Workshop on Live Programming (LIVE'13)*, pages 35–38. IEEE, 2013. (page 102)

[Toa] Serge Toarca. Debuggex. `http://www.debuggex.com/`. (page 101)

[TOHSJ99] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE'99)*, pages 107–119. ACM, 1999. (page 15)

[Vas11] Panos Vassiliadis. A Survey of Extract-Transform-Load Technology. In David Taniar and Li Chen, editors, *Integrations of Data Warehousing, Data Mining and Database Technologies - Innovative Approaches*, pages 171–199. Information Science Reference, 2011. (page 6)

[Vee07] Cor J. Veenman. Statistical Disk Cluster Classification for File Carving. In *Proceedings of the Third International Symposium on Information Assurance and Security (IAS'07)*, pages 393–398, 2007. (page 20)

[Vis97] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997. (page 55)

[W3C03] W3C. Portable Network Graphics (PNG) Specification, 2003. `http://www.w3.org/TR/PNG/`. (page 44, 94)

[Wil05] Svein Willassen. Forensic Analysis of Mobile Phone Internal Memory. In *Advances in Digital Forensics*, volume 194 of *IFIP – The International Federation for Information Processing*, pages 191–204. Springer, 2005. (page 7)

[XS05] Zhenchang Xing and Eleni Stroulia. UMLDiff: An Algorithm for Object-Oriented Design Differencing. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*, pages 54–65. ACM, 2005. (page 11, 89)

[Zim80] Hubert Zimmermann. OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, 28(4):425–432, April 1980. (page 13)

[ZL96] Andreas Zeller and Dorothea Lütkehaus. DDD - A Free Graphical Front-End for UNIX Debuggers. *SIGPLAN Notices*, 31(1):22–27, 1996. (page 101)

Summary

Digital forensics concerns the acquisition, recovery and analysis of information stored on digital devices for the purpose of answering legal questions. Exponential increases in available storage capacity and network bandwidth, as well as growing device and service adoption by the public, have made manual inspection of all potentially relevant information infeasible in nearly all cases. A solution to this problem is *automated digital forensics*, which is the use of software to perform tasks in digital forensics automatically, reducing the time required to get results.

Many software engineering techniques exist that allow the construction of high performance and scalable solutions in the domain of digital forensics. Unfortunately, another major requirement complicates the application of standard techniques: handling the high variability in the shape of how investigated information is stored. The number of different devices, networks, platforms, and applications is huge and constantly changing. This leads to a constant stream of required changes to digital forensics software in order to recover as much information as possible.

Factoring out the commonality so that the constantly changing aspects of a solution can evolve separately from the stable aspects is a supposed strength of *model-driven software engineering* (MDSE). This separation of concerns is achieved through the use of a domain-specific language (DSL), which is a custom notation used to specify the changing parts of a solution. Changes expressed in this DSL are then automatically applied through the use of transformation tools such as code generators and interpreters, which handle fixed requirements such as high performance.

This thesis presents analyses and experiments that were performed in order to discover the benefits and costs of applying model-driven software engineering, specifically in the development and maintenance of solutions in the domain of automated digital forensics. The contributions are the following:

- A description of the results of domain analyses to establish initial requirements, including the domain of automated digital forensics in general, and data recovery and aspects of binary file formats in particular. Specific areas of interest are identified for the development of binary file format validators.

- Design and implementation of a DSL to describe binary file formats, applied in a forensic data recovery tool called a *file carver*. Experimental evaluation shows that the proposed model-driven approach has no negative effects on the runtime performance and data recovery qualities of the final solution, but does allow clear separation of concerns and requires fewer lines of code to maintain.
- Application of model transformation to let the user of the file carver trade accuracy of data recovery for runtime performance, without requiring changes by a software engineer. Experimental evaluation on a custom benchmark shows that runtime performance gains of up to a factor of three can be achieved, at the expense of up to 8% in precision and 5% in recall.
- Design of an experimental approach to observe the maintenance characteristics of a DSL, by generating realistic maintenance scenarios from a corpus of representative inputs. Application of the approach to the proposed DSL shows that it can accomodate all expected changes, and also identifies three language features to consider for further improvement.
- Design and implementation of an integrated development environment (IDE) that provides the DSL user with a fully synchronized view of all relevant information during development and maintenance. This includes syntax-colored views of the static file format description, the dynamic data recovery program state, as well as the input data.

Finally, the research presented in this thesis forms an extensive case study in the application of MDSE in the domain of automated digital forensics, using the Rascal metaprogramming language. It provides concrete evidence for the successful application of MDSE in the domain of automated digital forensics, and contributes to knowledge about the application of MDSE in general. The concise and versatile implementations provide a strong case for the usefulness and applicability of Rascal in DSL engineering.

Samenvatting

Digitaal forensisch onderzoek is het veiligstellen, reconstrueren en analyseren van informatie opgeslagen op digitale gegevensdragers, met als doel het beantwoorden van juridische vragen. Exponentiële toename in opslagcapaciteit en netwerkbandbreedte, alsmede de groei in het gebruik van digitale apparaten en diensten, hebben ertoe geleid dat handmatige inspectie van alle potentieel relevante informatie onhaalbaar is geworden in vrijwel alle situaties. Een oplossing voor dit probleem ligt in *geautomatiseerde digitaal forensische techniek*, het gebruik van software voor het automatisch uitvoeren van grote hoeveelheden van het digitaal forensische onderzoek, waardoor sneller resultaten kunnen worden bereikt.

Er bestaan diverse softwaretechnieken om oplossingen te implementeren die voldoen aan de eisen voor snelheid en schaalbaarheid in digitaal forensisch onderzoek. Helaas is er nog een belangrijke eis die de toepassing van deze technieken bemoeilijkt: het omgaan met de hoge variabiliteit van manieren waarop informatie wordt opgeslagen. De hoeveelheid verschillende apparaten, netwerken, platforms en toepassingen is zeer groot en continu in beweging. Dit leidt tot een constante stroom van vereiste aanpassingen aan digitaal forensische software, om ervoor te zorgen dat er zo veel mogelijk sporen worden gevonden.

Het loskoppelen van de veranderende aspecten van een oplossing, zodat deze onafhankelijk van de stabiele onderdelen kunnen evolueren, is een veronderstelde kracht van *model-gedreven software ontwikkeling*. Deze scheiding van aandachtsgebieden wordt gerealiseerd door gebruik van een domein-specifieke taal, een op maat gemaakte notatie om de frequent wijzigende aspecten van een oplossing te beschrijven. Veranderingen uitgedrukt in deze taal worden automatisch doorgevoerd door gebruik van transformatoren zoals code generatoren, die zorg dragen voor vaste eisen zoals snelheid en schaalbaarheid.

Dit proefschrift presenteert analyses en experimenten die zijn uitgevoerd om de voordelen en kosten van het toepassen van model-gedreven software ontwikkeling te ontdekken, specifiek op het gebied van ontwikkeling en onderhoud van geautomatiseerde digitaal forensische techniek. De bijdragen zijn als volgt:

- Een beschrijving van de resultaten van domeinanalyses om technische eisen vast te stellen, op het gebied van geautomatiseerde digitaal forensische techniek, datareconstructie en aspecten van binaire bestandsformaten.
- Ontwerp en implementatie van een domein-specifieke taal om binaire bestandsformaten te beschrijven, inclusief toepassing ervan in een forensische datareconstructie-applicatie (een zogenaamde *file carver*). Experimentele evaluatie laat zien dat de voorgestelde model-gedreven aanpak geen negatieve gevolgen heeft voor de resultaten op het gebied van snelheid en datareconstructie, maar dat het wel leidt tot een duidelijke scheiding van aandachtsgebieden en minder regels code om te onderhouden.
- Toepassing van modeltransformatie om de gebruiker de mogelijkheid te geven om precisie van datareconstructie te ruilen voor snelheid, zonder dat een software ontwikkelaar nodig is. Experimentele evaluatie op een op maat gemaakte benchmark laat zien dat de snelheid met een factor drie kan toenemen, in ruil voor een verlaging van de precisie met 8% en recall met 5%.
- Ontwerp van een experimentele aanpak om de onderhoudseigenschappen van een domein-specifieke taal te observeren, door realistische onderhoudsscenario's te genereren uit een corpus van representatieve invoeren. Toepassing van deze aanpak op de ontwikkelde domein-specifieke taal laat zien dat deze in staat is om al het te verwachten onderhoud te verwerken en welke functies meerwaarde kunnen hebben als toevoeging aan de taal.
- Ontwerp en ontwikkeling van een geïntegreerde ontwikkelomgeving die de gebruiker van de domein-specifieke taal een volledig gesynchroniseerd beeld geeft van alle relevante informatie tijdens het ontwikkelen en onderhouden van forensische software. Dit gesynchroniseerde beeld omvat de statische bestandsformaatbeschrijving, de dynamische staat van het datareconstructieprogramma en de invoerdata.

Tenslotte, het onderzoek in dit proefschrift vormt een uitgebreide case study in de toepassing van model-gedreven software ontwikkeling in het domein van geautomatiseerde digitaal forensische techniek, op basis van de Rascal metaprogrammeertaal. Het beschrijft concreet bewijs voor de succesvolle toepassing van model-gedreven software ontwikkeling in geautomatiseerde digitaal forensische techniek en draagt bij aan kennis over toepassing van model-gedreven software ontwikkeling in het algemeen. De compacte en flexibele implementaties laten zien dat Rascal uitermate geschikt is voor de toepassing van model-gedreven software ontwikkeling.

Titles in the IPA Dissertation Series since 2008

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Process-*

ing Systems. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean*. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange*. Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web*. Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*. Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis*. Fac-

ulty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks*. Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement*. Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers*. Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics*. Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics*. Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software*. Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra*. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications*. Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services*. Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates*. Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement*. Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proenc ca. *Synchronous coordination of distributed components*. Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Moralı. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors*. Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory*. Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model co-evolution*. Faculty of Mathematics and Computer Science, TU/e. 2011-12

S. Georgievska. *Probability and Hiding in Concurrent Processes*. Faculty of Mathematics and Computer Science, TU/e. 2011-13

S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

M. Raffelsieper. *Cell Libraries and Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-15

C.P. Tsirogiannis. *Analysis of Flow and Visibility on Triangulated Terrains*. Faculty of Mathematics and Computer Science, TU/e. 2011-16

Y.-J. Moon. *Stochastic Models for Quality of Service of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-17

R. Middelkoop. *Capturing and Exploiting Abstract Views of States in OO Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-18

M.F. van Amstel. *Assessing and Improving the Quality of Model Transformations*. Faculty of Mathematics and Computer Science, TU/e. 2011-19

A.N. Tamalet. *Towards Correct Programs in Practice*. Faculty of Science,

Mathematics and Computer Science, RU. 2011-20

H.J.S. Basten. *Ambiguity Detection for Programming Language Grammars*. Faculty of Science, UvA. 2011-21

M. Izadi. *Model Checking of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-22

L.C.L. Kats. *Building Blocks for Language Workbenches*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23

S. Kemper. *Modelling and Analysis of Real-Time Coordination Patterns*. Faculty of Mathematics and Natural Sciences, UL. 2011-24

J. Wang. *Spiking Neural P Systems*. Faculty of Mathematics and Natural Sciences, UL. 2011-25

A. Khosravi. *Optimal Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2012-01

A. Middelkoop. *Inference of Program Properties with Attribute Grammars, Revisited*. Faculty of Science, UU. 2012-02

Z. Hemel. *Methods and Techniques for the Design and Implementation of Domain-Specific Languages*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03

T. Dimkov. *Alignment of Organizational Security Policies: Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04

S. Sedghi. *Towards Provably Secure Efficiently Searchable Encryption*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05

F. Heidarian Dehkordi. *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference*. Faculty of Science, Mathematics and Computer Science, RU. 2012-06

K. Verbeek. *Algorithms for Cartographic Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2012-07

D.E. Nadales Agut. *A Compositional Interchange Format for Hybrid Systems: Design and Implementation*. Faculty of Mechanical Engineering, TU/e. 2012-08

H. Rahmani. *Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2012-09

S.D. Vermolen. *Software Language Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10

L.J.P. Engelen. *From Napkin Sketches to Reliable Software*. Faculty of Mathematics and Computer Science, TU/e. 2012-11

F.P.M. Stappers. *Bridging Formal Models – An Engineering Perspective*. Faculty of Mathematics and Computer Science, TU/e. 2012-12

W. Heijstek. *Software Architecture Design in Global and Model-Centric Software Development*. Faculty of Mathematics and Natural Sciences, UL. 2012-13

C. Kop. *Higher Order Termination*. Faculty of Sciences, Department of Computer Science, VUA. 2012-14

A. Osaiweran. *Formal Development of Control Software in the Medical Systems Domain*. Faculty of Mathematics and Computer Science, TU/e. 2012-15

W. Kuijper. *Compositional Synthesis of Safety Controllers*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-16

H. Beohar. *Refinement of Communication and States in Models of Embedded Systems*. Faculty of Mathematics and Computer Science, TU/e. 2013-01

G. Igna. *Performance Analysis of Real-Time Task Systems using Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2013-02

E. Zambon. *Abstract Graph Transformation – Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-03

B. Lijnse. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2013-04

G.T. de Koning Gans. *Outsmarting Smart Cards*. Faculty of Science, Mathematics and Computer Science, RU. 2013-05

M.S. Greiler. *Test Suite Comprehension for Modular and Dynamic Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-06

L.E. Mamane. *Interactive mathematical documents: creation and presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2013-07

M.M.H.P. van den Heuvel. *Composition and synchronization of real-time components upon one processor*. Faculty of Mathematics and Computer Science, TU/e. 2013-08

J. Businge. *Co-evolution of the Eclipse Framework and its Third-party Plug-ins*. Faculty of Mathematics and Computer Science, TU/e. 2013-09

S. van der Burg. *A Reference Architecture for Distributed Software Deployment*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-10

J.J.A. Keiren. *Advanced Reduction Techniques for Model Checking*. Faculty of Mathematics and Computer Science, TU/e. 2013-11

D.H.P. Gerrits. *Pushing and Pulling: Computing push plans for disk-shaped robots, and dynamic labelings for moving points*. Faculty of Mathematics and Computer Science, TU/e. 2013-12

M. Timmer. *Efficient Modelling, Generation and Analysis of Markov Automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-13

M.J.M. Roeloffzen. *Kinetic Data Structures in the Black-Box Model*. Faculty of Mathematics and Computer Science, TU/e. 2013-14

L. Lensink. *Applying Formal Methods in Software Development*. Faculty of Science, Mathematics and Computer Science, RU. 2013-15

C. Tankink. *Documentation and Formal Mathematics — Web Technology meets Proof Assistants*. Faculty of Science, Mathematics and Computer Science, RU. 2013-16

J. van den Bos. *Gathering Evidence: Model-Driven Software Engineering in Automated Digital Forensics*. Faculty of Science, UvA. 2014-01