A Case Study in Evidence-Based DSL Evolution

Jeroen van den Bos[1,2] and Tijs van der Storm[1]

[1] Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
[2] Netherlands Forensic Institute, Den Haag, The Netherlands
jeroen@infuse.org, storm@cwi.nl

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.

1 Introduction

Domain-specific languages (DSLs) can increase productivity by trading generality for expressive power [17, 5]. 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 [4]. How can we find out whether the relevant maintenance scenarios will require routine changes or not?

In this paper we present a test-based experimental approach to answer this question and apply it to a domain-specific language for describing file formats: DERRIC [2]. 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

```
format PNG
extension png
strings ascii
sequence Signature IHDR Chunk* IEND

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

Chunk {
  length: lengthOf(chunkdata) size 4;
  chunktype: type string size 4;
  chunkdata: size length;
  crc: checksum(
         algorithm="crc32-ieee",
         init="allone",start="lsb",
         end="invert",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;
  }
}

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

Fig. 1. Simplified PNG in DERRIC.

expressed more easily. Thus, the experiment has provided us with empirical data to improve the design of DERRIC.

The contributions of this paper 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.

2 Background

DERRIC is a DSL to describe binary file formats [2]. It is used in digital forensics investigations to construct highly flexible and high performance analysis tools. One example is the construction of file carvers [1], 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. 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.

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 paper, we focus on *corrective* maintenance [10], 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 [19]. 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 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[3]. 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 to qualitatively assess to what extent DERRIC supports maintenance of format descriptions.

[3] 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.

4 Experiment

4.1 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 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 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 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 [3]. This data set contains JPEG, GIF and PNG images found on Wikipedia, downloaded using the latest available static dump list, which dates from 2008[4]. 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 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. 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

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

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.

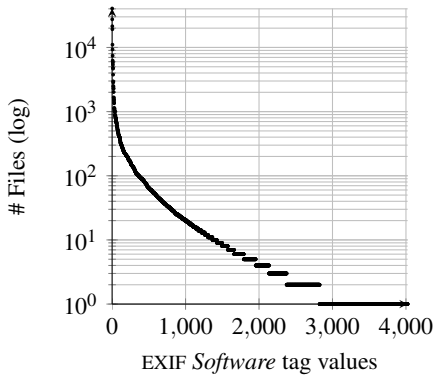

Fig. 2. Distribution of EXIF *Software* tag values over 28.4% of the corpus

4.2 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.

4.3 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 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

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

sequence expression (cf. Fig. 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[5] to allow full traceability and reproducability of the results of this paper. In fact, a single change corresponds to a single commit. After each change the DERRIC compiler was rerun with the modified descriptions. The process was repeated until all failures were resolved.

4.4 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 [7] 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.

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

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

5 Results

The results of the experiment are summarized in Table 4, 5 and 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 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.

6 Analysis

To summarize the results of our experiment, Table 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 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.

Id	Structure Add	Structure Mod	Sequence Add	Sequence Mod	Errors 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 4. Modifications to the JPEG description.

Id	Structure Add	Structure Mod	Sequence Add	Sequence Mod	Errors 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. Modifications to the GIF description.

Id	Structure Add	Structure Mod	Sequence Add	Sequence Mod	Errors 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 6. Modifications to the PNG description.

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

Table 7. Changes per change complexity class

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)*
```

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`.

7 Discussion

7.1 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 [14] 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 [20]. Generic difference algorithms (on trees or graphs) would likely contain irrelevant noise, and hence would be hard to interpret.

7.2 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 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. 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 4 we have argued that the DERRIC descriptions of these file formats are sufficiently different to cover the whole language.

7.3 Related Work

Mens et al. [16] 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. [9] 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 paper are relatively small, writing them as *actual* metaprograms might be practically feasible. Even more so since DERRIC is implemented using the metaprogramming language RASCAL [11], which is highly suitable for expressing the changes as source-to-source transformations.

The work presented in this paper can be positioned as an experiment in language evaluation. Empirical language evaluation is relatively new since, as pointed out by Markstrum [15], 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 [17], key success factors [8], and maintainability [12]. 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 [6] for a comprehensive list of references. A recent study is performed by Lämmel and Pek. [13]. 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. [18] 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 paper 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.

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 paper 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 paper as a first step towards evidence-based DSL evolution.

References

1. Aronson, L., van den Bos, J.: Towards an Engineering Approach to File Carver Construction. In: 2011 IEEE 35th Annual Computer Software and Applications Conference Workshops (COMPSACW). pp. 368–373. IEEE (2011)
2. van den Bos, J., van der Storm, T.: Bringing Domain-Specific Languages to Digital Forensics. In: 33rd International Conference on Software Engineering (ICSE'11). pp. 671–680. ACM (2011)
3. van den Bos, J., van der Storm, T.: Domain-Specific Optimization in Digital Forensics. In: Hu, Z., de Lara, J. (eds.) 5th International Conference on Model Transformation (ICMT'12). LNCS, vol. 7307, pp. 121–136. Springer (2012)
4. van Deursen, A., Klint, P.: Little Languages: Little Maintenance? Journal of Software Maintenance 10(2), 75–92 (1998)
5. van Deursen, A., Klint, P., Visser, J.: Domain-Specific Languages: An Annotated Bibliography. SIGPLAN Notices 35(6), 26–36 (2000)
6. Favre, J.M., Gasevic, D., Lämmel, R., Pek, E.: Empirical Language Analysis in Software Linguistics. In: Malloy, B.A., Staab, S., van den Brand, M. (eds.) Third International Conference on Software Language Engineering (SLE'10). LNCS, vol. 6563, pp. 316–326. Springer (2010)
7. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring. Addison-Wesley (1999)
8. Hermans, F., Pinzger, M., van Deursen, A.: Domain-Specific Languages in Practice: A User Study on the Success Factors. In: Schürr, A., Selic, B. (eds.) 12th International Conference On Model Driven Engineering Languages And Systems (MODELS'09). LNCS, vol. 5795, pp. 423–437. Springer (2009)
9. Hills, M., Klint, P., van der Storm, T., Vinju, J.J.: A Case of Visitor versus Interpreter Pattern. In: Bishop, J., Vallecillo, A. (eds.) 49th Int. Conference on Objects, Models, Components and Patterns (TOOLS'11). Lecture Notes in Computer Science, vol. 6705, pp. 228–243. Springer (2011)
10. ISO/IEC 14764: Software Engineering–Software Life Cycle Processes–Maintenance (2006)
11. Klint, P., van der Storm, T., Vinju, J.: Rascal: A Domain Specific Language for Source Code Analysis and Manipulation. In: Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'09). pp. 168–177. IEEE (2009)
12. Klint, P., van der Storm, T., Vinju, J.J.: 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)
13. Lämmel, R., Pek, E.: 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). pp. 104–113. IEEE (2010)

14. Lussenburg, V., van der Storm, T., Vinju, J.J., Warmer, J.: Mod4J: A Qualitative Case Study of Model-Driven Software Development. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) 13th International Conference on Model Driven Engineering Languages and Systems (MODELS'10) Part II. LNCS, vol. 6395, pp. 346–360. Springer (2010)
15. Markstrum, S.: 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). pp. 7:1–7:5. ACM (2010)
16. Mens, T., Eden, A.H.: On the Evolution Complexity of Design Patterns. Electronic Notes in Theoretical Computer Science 127(3), 147–163 (2005)
17. Mernik, M., Heering, J., Sloane, A.M.: When and How to Develop Domain-Specific Languages. ACM Computing Surveys 37(4), 316–344 (2005)
18. Morandat, F., Hill, B., Osvald, L., Vitek, J.: Evaluating the Design of the R Language - Objects and Functions for Data Analysis. In: Noble, J. (ed.) 26th European Conference on Object-Oriented Programming (ECOOP'12). LNCS, vol. 7313, pp. 104–131. Springer (2012)
19. Oehlert, P.: Violating Assumptions with Fuzzing. IEEE Security and Privacy 3(2), 58–62 (2005)
20. Xing, Z., Stroulia, E.: UMLDiff: An Algorithm for Object-Oriented Design Differencing. In: 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05). pp. 54–65. ACM (2005)