Origin Tracking + Text Differencing = Textual Model Differencing

Riemer van Rozen[1] and Tijs van der Storm[2,3]

[1] Amsterdam University of Applied Sciences
[2] Centrum Wiskunde & Informatica
[3] Universiteit van Amsterdam

Abstract. In textual modeling, models are created through an intermediate parsing step which maps textual representations to abstract model structures. Therefore, the identify of elements is not stable across different versions of the same model. Existing model differencing algorithms, therefore, cannot be applied directly because they need to identify model elements across versions. In this paper we present Textual Model Diff (TMDIFF), a technique to support model differencing for textual languages. TMDIFF requires origin tracking during text-to-model mapping to trace model elements back to the symbolic names that define them in the textual representation. Based on textual alignment of those names, TMDIFF can then determine which elements are the same across revisions, and which are added or removed. As a result, TMDIFF brings the benefits of model differencing to textual languages.

1 Introduction

Model differencing algorithms (e.g., [1]) determine which elements are added, removed or changed between revisions of a model. A crucial aspect of such algorithms that model elements need to be identified across versions. This allows the algorithm to determine which elements are still the same in both versions. In textual modeling [6], models are represented as textual source code, similar to Domain-Specific Languages (DSLs) and programming languages. The actual model structure is not first-class, but is derived from the text by a text-to-model mapping, which, apart from parsing the text into a containment hierarchy also provides for reference resolution. After every change to the text, the corresponding structure needs to be derived again. As a result, the identities assigned to the model elements during text-to-model mapping are not preserved across versions, and model differencing cannot be applied directly.

Existing approaches to textual model differencing are based on mapping textual syntax to a standard model representation (e.g., languages built with Xtext are mapped to EMF [5]) and then using standard model comparison tools (e.g., EMFCompare [2, 3]). As a result, model elements in both versions are matched using name-based identities stored in the model elements themselves. One approach is to interpret such names as globally unique identifiers: match model elements of the same class, irrespective of their location in the containment hierarchy of the model. Another approach is to only match elements in collections at the same position in the containment hierarchy.

Unfortunately, both approaches have their limitations. In the case of global names, the language cannot have scoping rules: it is impossible to have different model elements of the same class with the same name. On the other hand, matching names relative to the containment hierarchy entails that scoping rules must obey the containment hierarchy, which limits flexibility.

In this paper we present TMDIFF, a language-parametric technique for model differencing of textual languages which does support languages with complex scoping rules, but at the same time is agnostic of the model containment hierarchy. As a result, different elements with the same name, but in different scopes can still be identified. TMDIFF is based on two key techniques:

– **Origin tracking.** In order to map model element identities back to the source, we assume that the text-to-model mapping applies origin tracking [7, 19]. Origin tracking induces an *origin relation* which relates source locations of definitions to (opaque) model identities. Each semantic model element can be traced back to its defining name in the textual source, and each defining name can be traced forward to its corresponding model element.
– **Text Differencing.** TMDIFF identifies model elements by textually aligning definition names between two versions of a model using traditional text differencing techniques (e.g., [11]). When two names in the textual representations of two models are aligned, they are assumed to represent the "same" model element in both models. In combination with the origin relation this allows TMDIFF to identify the corresponding model elements as well.

The resulting identification of model elements can be passed to standard model differencing algorithms, such as the one by Alanen and Porres [1].

TMDIFF enjoys the important benefit that it is fully language parametric. TMDIFF works irrespective of the specific binding semantics and scoping rules of a textual modeling language. In other words, how the textual representation is mapped to model structure is irrelevant. The only requirement is that semantic model elements are introduced using symbolic names, and that the text-to-model mapping performs origin tracking.

The contributions of this paper are summarized as follows:

– We explore how textual differencing can be used to match model elements based on origin tracking information.
– We provide a detailed description of TMDIFF, including a prototype implementation.
– The feasibility of the approach is illustrated by applying TMDIFF in the context of a realistic, independently developed DSL.

2 Overview

Here we introduce textual model differencing using a simple motivating example that is used as a running example throughout the paper. Figure 1 shows a state machine model for controlling doors. It is both represented as text (left) and as object diagram (right). A state machine has a name and contains a number of state declarations. Each state declaration contains zero or more transitions. A transition fires on an event, and then transfers control to a new state.

```
machine doors d1
  state closed d2
    open => opened

  state opened d3
    close => closed
end
```

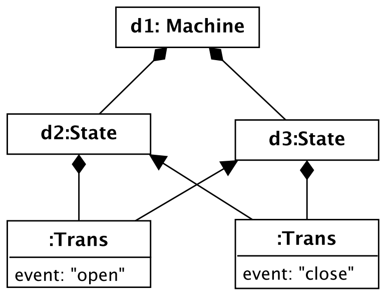

Fig. 1: *Doors*$_1$: a simple textual representation of a state machine and its model.

The symbolic names that *define* entities are annotated with unique labels d_n. These labels capture *source locations* of names. That is, a name occurrence is identified with its line and column number and/or character offset[4]. Since identifiers can never overlap, labels are guaranteed to be unique, and the actual name corresponding to label can be easily retrieved from the source text itself. For instance, the machine itself is labeled d_1, and both states `closed` and `opened` are labeled d_2 and d_3 respectively.

The labels are typically the result of *name analysis* (or reference resolution), which distinguishes definition occurrences of names from use occurrences of names according to the specific scoping rules of the language. For the purpose of this paper it is immaterial how this name analysis is implemented, or what kind of scoping rules are applied. The important aspect is to know which name occurrences represent definitions of elements in the model.

By propagating the source locations (d_i) to the fully resolved model, symbolic names can be linked to model elements and vice versa. On the right of Fig. 1, we have used the labels themselves as object identities in the object model. Note that the anonymous Transition objects lack such labels. In this case, the objects do not have an identity, and the difference algorithm will perform structural differencing (e.g., [20]), instead of semantic, model-based differencing [1].

Figure 2 shows two additional versions of the state machine of Fig. 1. First the machine is extended with a `locked` state in *Doors*$_2$ (Fig. 2a). Second, *Doors*$_3$ (Fig. 2c), shows a grouping feature of the language: the `locked` state is part of the `locking` group. The grouping construct acts as a scope: it allows different states with the same name to coexist in the same state machine model.

Looking at the labels in Fig. 1 and 2, however, one may observe that the labels used in each version are disjoint. For instance, even though the defining name occurrences of the machine `doors` and state `closed` occur at the exact same location in *Doors*$_2$ and *Doors*$_3$, this is an accidental artifact of how the source code is formatted. Case in point is the name `locked`, which now has moved down because of the addition of the group construct.

[4] For the sake of presentation, we use the abstract labels d_i for the rest of the paper, but keep in mind that they represent source locations

```
machine doors d4
  state closed d5
    open => opened
    lock => locked

  state opened d6
    close => closed

  state locked d7
    unlock => closed

end
```

(a) *Doors*$_2$

```
machine doors d8
  state closed d9
    open => opened
    lock => locking.locked

  state opened d10
    close => closed

  locking d11 {
    state locked d12
      unlock => closed
  }
end
```

(b) *Doors*$_3$

Fig. 2: Two new versions of the simple state machine model *Doors*$_1$.

$$\begin{tikzcd}
src_1 \arrow[r, "\mathsf{map}"'] \arrow[d, dashed, "\Delta"] \arrow[d, leftrightarrow, bend right, "\mathsf{align}"'] & m_1 \arrow[l, dotted, bend right, "origin_1"'] \arrow[d, leftrightarrow, dotted, "\mathsf{identify}"] \\
src_2 \arrow[r, "\mathsf{map}"'] & m_2 \arrow[l, dotted, bend right, "origin_2"']
\end{tikzcd}$$

Fig. 3: Identifying model elements in m_1 and m_2 through origin tracking and alignment of textual names.

The source locations, therefore, cannot be used as (stable) identities to used during model differencing. The approach taken by TMDIFF involves determining added and removed definitions by aligning the textual occurrences of defining names (i.e. labels d_i). Based on the origin tracking between the textual source and the actual model it then becomes possible to identify which model elements have survived changing the source text.

This high-level approach is visualized in Fig. 3. src_1 and src_2 represent the source code of two revisions of a model. Each of these textual representations is mapped to a proper model, m_1 and m_2 respectively. Mapping text to a model induces origin relations, $origin_1$ and $origin_2$, mapping model elements back to the source locations of their defining names in src_1 and src_2 respectively. By then aligning these names between src_1 and src_2, the elements themselves can be identified via the respective origin relations.

```
--- a/doors1.sl
+++ b/doors2.sl
@@ -3,0 +4
+    lock => locked
@@ -6,0 +8,3
+
+  state locked
+    unlock => closed
```

```
--- a/doors2.sl
+++ b/doors3.sl
@@ -4 +4
-    lock => locked
+    lock => locking.locked
@@ -8,0 +9
+  locking {
@@ -10,0 +12
+   }
```

Fig. 4: Textual diff between $Doors_1$ and $Doors_2$, and $Doors_2$ and $Doors_3$[5].

```
create State d7
d7 = State("locked",[Trans("unlock",d2)])
d2.out[1] = Trans("lock", d7)
d1.states[2] = d7
```

(a) `tmdiff` $Doors_1$ $Doors_2$

```
create Group d11
d11 = Group("locking",[d7])
remove d4.states[2]
d4.states[2] = d11
```

(b) `tmdiff` $Doors_2$ $Doors_3$

Fig. 5: TMDIFF differences between $Doors_i$ and $Doors_{i+1}$ ($i \in 1,..,2$)

TMDIFF aligns textual names by interpreting the output of a textual `diff` algorithm on the model source code. The diffs between $Doors_1$ and $Doors_2$, and $Doors_2$ and $Doors_3$ is shown in Fig. 4. As can be seen, the diffs show for each line whether it was added ("+") or removed ("-"). By looking at the line number of the definition labels d_i it becomes possible to determine whether the associated model element was added or removed.

For instance, the new `locked` state was introduced in $Doors_2$. This can be observed from the fact that the diff on the left of Fig. 4 shows that the name "locked" is on a line marked as added. Since the names `doors`, `closed` and `opened` occur on unchanged lines, TMDIFF will identify the corresponding model elements (the machine, and the 2 states) in $Doors_1$ and $Doors_2$. Similarly, the diff between $Doors_2$ and $Doors_3$ shows that only the group `locking` was introduced. All other entities have remained the same, even the `locked` state, which has moved into the group `locking`.

With the identification of model elements in place, TMDIFF applies a variant of the standard model differencing introduced in [1]. Hence, TMDIFF deltas are imperative edit scripts that consist of edit operations on the model. Edit operations include creating and removing of nodes, assigning fields, and inserting or removing elements from collection-valued properties. Figure 5 shows the TMDIFF edit scripts computed between $Doors_1$ and $Doors_2$ (a), and $Doors_2$ and $Doors_3$ (b). The edit scripts use the definition labels d_n as node identities.

The edit script shown in Fig. 5a captures the difference between source version $Doors_1$ and target version $Doors_2$. It begins with the creation of a new state d_7. On the following line d_7 is initialized with its name (`locked`) and a fresh collection of transitions.

[5] The diffs are computed by the `diff` tool included with the `git` version control system. We used the following invocation: `git diff --no-index --patience --ignore-space-change --ignore-blank-lines --ignore-space-at-eol -U0 <old> <new>`.

```
list[Operation] tmDiff(str $src_1$, str $src_2$, obj $m_1$, obj $m_2$) {
 <$A$, $D$, $M$> = match($src_1$, $src_2$, $m_1$, $m_2$)
 $\Delta$ = [ new Create($d_a$, $d_a$.class) | $d_a \leftarrow A$ ]
 $M' = M$ + { <$d_a$, $d_a$> | $d_a \leftarrow A$ }
 $\Delta$ += [ new SetTree($d_a$, build($d_a$, $M'$)) | $d_a \leftarrow A$ ]
 for (<$d_1$, $d_2$> $\leftarrow M$)
    $\Delta$ += diffNodes($d_1$, $d_1$, $d_2$, [], $M'$)
 $\Delta$ += [ new Delete($d_d$) | $d_d \leftarrow D$ ]
 return $\Delta$
}
```

Fig. 6: TMDIFF

The transitions are *contained* by the state, so they are created anonymously (without identity). Note that the created transition contains a (cross-)reference to state d_2. The next step is to add a new transition to the `out` field of state d_2 (which is preserved from $Doors_1$). The target state of this transition is the new state d_7. Finally, state d_7 is inserted at index 2 of the collection of states of the machine d_1 in $Doors_1$.

The edit script introducing the grouping construct `locking` between $Doors_2$ and $Doors_3$ is shown in Fig. 5b. The first step is the creation of a new group d_{11}. It is initialized with the name `"locking"`. The set of nested states is initialized to contain state d_7 which already existed in $Doors_2$. Finally, the state with index 2 is removed from the machine d_4 in $Doors_3$, and then replaced by the new group d_{11}.

In this section we have introduced the basic approach of TMDIFF using the state machine example. The next section presents TMDIFF in more detail.

3 TMDIFF in More Detail

3.1 Top-level Algorithm

Figure 6 shows the TMDIFF algorithm in high-level pseudo code. Input to the algorithm are the source texts of the models (src_1, src_2), and the models themselves (m_1, m_2). The first step is identifying model elements of m_1 to elements in m_2 using the matching technique introduced above. The `match` function is further described in the next sub section (Section 3.2).

Based on the matching returned by `match`, TMDIFF first generates global Create operations for nodes that are in the A set. After these operations are created, the matching M is "completed" into M', by mapping every added object to itself. This ensures that reverse lookups in M' for elements in m_2 will always be defined. Each entity just created is initialized by generating SetTree operations which reconstruct the containment hierarchy for each element d_a using the build function. The function diffNodes then computes the difference between each pair of nodes originally identified in M. The edit operations will be anchored at object d_1 (first argument). As a result, diffNodes produces edits on "old" entities, if possible. Finally, the nodes that have been deleted from m_1 result in global Delete actions.

```
Matching match(str $src_1$, str $src_2$, obj $m_1$, obj $m_2$) {
  $P_1$ = project($m_1$)
  $P_2$ = project($m_2$)
  $<L_{add}, L_{del}>$ = split(diff($src_1$, $src_2$))

  $i$ = 0, $j$ = 0; $A = \{\}$, $D = \{\}$; $M = \{\}$
  while ($i < |P_1| \vee j < |P_2|$) {
      if ($i < |P_1| \wedge P_1[i].line \in L_{del}$)
        $D$ += $\{P_1[i].object\}$; $i$ += 1; continue
      if ($j < |P_2| \wedge P_2[j].line) \in L_{add}$)
        $A$ += $\{P_2[j].object\}$; $j$ += 1; continue
      if ($P_1[i].object.class = P_2[j].object.class$)
        $M$ += $\{<P_1[i].object, P_2[j].object>\}$
      else
        $D$ += $\{P_1[i].object\}$; $A$ += $\{P_2[j].object\}$
      $i$ += 1; $j$ += 1
  }
  return $<A, D, M>$;
}
```

Fig. 7: Matching model elements based on source text diffs.

3.2 Matching

The match function uses the output computed by standard `diff` tools. In particular, we employ a `diff` variant called *Patience Diff*[6] which is known to often provide better results than the standard, LCS-based, algorithm [12].

The matching algorithm is shown in Fig. 7. The function match takes the textual source of both models (src_1, src_2) and the actual models as input (m_1, m_2). It first projects out the origin and class information for each model. The resulting projections P_1 and P_2 are sequences of tuples $\langle x, c, l, d\rangle$, where x is the symbolic name of the entity, c its class (e.g. State, Machine, etc.), l the textual line it occurs on and d the object itself.

As an example, the projections for $Doors_1$ and $Doors_2$ are as follows:

$$P_1 = \begin{array}{l} [\ \langle \texttt{doors}, \ \textit{Machine}, 1, d_1\rangle, \\ \langle \texttt{closed}, \textit{State}, \quad 2, d_2\rangle, \\ \langle \texttt{opened}, \textit{State}, \quad 5, d_3\rangle\ ] \end{array} \qquad P_2 = \begin{array}{l} [\ \langle \texttt{doors}, \ \textit{Machine}, 1, d_4\rangle, \\ \langle \texttt{closed}, \textit{State}, \quad 2, d_5\rangle, \\ \langle \texttt{opened}, \textit{State}, \quad 6, d_6\rangle, \\ \langle \texttt{locked}, \textit{State}, \quad 9, d_7\rangle\ ] \end{array}$$

The algorithm then partitions the textual `diff` in two sets L_{add} and L_{del} of added lines (relative to src_2) and deleted lines (relative to src_1). The main **while**-loop then iterates over the projections P_1 and P_2 in parallel, distributing definition labels over the A, D and M sets that will make up the matching. If a name occurs unchanged in both src_1 and src_2, an additional type check prevents that entities in different categories are matched.

[6] See: `http://bramcohen.livejournal.com/73318.html`

The result of matching is a triple $M = \langle A, D, I\rangle$, where $A \subseteq L_Y$ contains new elements in Y, $D \subseteq L_X$ contains elements removed from X, and $I \subseteq L_X \times L_Y$ represents identified entities.

For instance the matchings between $Doors_1$, $Doors_2$, and between $Doors_2$ and $Doors_3$ are:

$$M_{1,2} = \langle\{d_7\},\{\},\{\langle d_1,d_4\rangle,\langle d_2,d_5\rangle,\langle d_3,d_6\rangle\}\rangle$$
$$M_{2,3} = \langle\{d_{11}\},\{\},\{\langle d_4,d_8\rangle,\langle d_5,d_9\rangle,\langle d_6,d_{10}\rangle,\langle d_7,d_{12}\rangle\}\rangle$$

3.3 Differencing

```
list[Operation] diffNodes(obj $ctx$, obj $t_1$, obj $t_2$, Path $p$, Matching $M$) {
  assert $t_1$.class = $t_2$.class;
  $\Delta$ = []
  for ($f \leftarrow m_1$.class.fields) {
    if ($f$.isPrimitive && $t_1[f] \neq t_2[f]$)
      $\Delta$ += [new SetPrim($ctx$, $p + [f]$, $t_2[f]$)];
    else if ($f$.isContainment)
      if ($m_1[f]$.class = $m_2[f]$.class)
        $\Delta$ += diffNodes($ctx$, $t_1[f]$, $t_2[f]$, $p + [f]$, $M$)
      else
        $\Delta$ += [new SetTree($ctx$, $p + [f]$, build($m_2[f]$, $M$))]
    else if ($f$.isReference && $\boxed{M^{-1}[t_2[f]] \neq t_1[f]}$)
      $\Delta$ += [new SetRef($ctx$, $p + [f]$, $\boxed{M^{-1}[t_2[f]]}$)]
    else if ($f$.isList)
      $\Delta$ += diffLists($ctx$, $t_1[f]$, $t_2[f]$, $p + [f]$, $M$)
  }
  return $\Delta$
}
```

Fig. 8: Differencing nodes.

The heavy lifting of TMDIFF is realized by the diffNodes function. It is shown in Fig. 8. It receives the current context (ctx), the two elements to be compared (t_1 and t_2), a Path p which is a list recursively built up out of names and indexes and the matching relation to provide reference equality between elements in t_1 and t_2. diffNodes assumes that both t_1 and t_2 are of the same class. The algorithm then loops over all fields that need to be differenced. Fields can be of four kinds: primitive, containment, reference or list. For each case the appropriate edit operations are generated, and in most cases the semantics is straightforward and standard. For instance, if the field is list-valued, we delegate differencing to an auxiliary function diffLists (not shown) which performs Longest Common Subsequence (LCS) differencing using reference equality. The interesting bit happens when differencing reference fields. References are compared via the matching M. Figure 8 highlights the relevant parts.

In order to know whether two references are "equal", diffNodes performs a reverse lookup in M on the reference in t_2. If the result of that lookup is different from the reference in t_1 the field needs to be updated. Recall that M was augmented to M' (cf. Fig. 6) to contain entries for all newly created model elements. As a result, the reverse lookup is always well-defined. Either we find an already existing element of t_1, or we find a element created as part of t_2.

4 Case study: Derric

4.1 Implementation in RASCAL

We have implemented TMDIFF in RASCAL, a functional programming language for meta programming and a language workbench for developing textual Domain-Specific Languages (DSLs) [8]. The code for the algorithm, and the application to the example state machine language and the case study can be found on GitHub[7].

Since RASCAL is a textual language workbench [4] all models are represented as text, and then parsed into an abstract syntax tree (AST). Except for primitive values (string, boolean, integer etc.), all nodes in the AST are automatically annotated with source locations to provide basic origin tracking.

Source locations are a built-in data type in RASCAL (**loc**), and are used to relate sub-trees of a parse tree or AST back to their corresponding textual source fragment. A source location consists of a resource URI, an offset, a length, and begin/end and line/column information. For instance, the name of the `closed` state in Fig. 2 is labeled:

```
|project://textual-model-diff/input/doors1.sl|(22,6,<2,8>,<2,14>)
```

Because RASCAL is a functional programming language, all data is immutable. As a result graph-like structure cannot be directly represented. Instead we represent the containment hierarchy of a model as an AST, and represent cross-references by explicit relations **rel**[**loc** from, **loc** to], once again using source locations to represent object identities.

4.2 Differencing Derric File Format Descriptions

To evaluate TMDIFF on a real-life DSL and see if it computes reasonable deltas, we have applied it to the version history of *file format specifications*. These file format specifications are written in Derric, a DSL for digital forensics analysis [16]. Derric is a grammar-like DSL: it contains a top-level regular expression, specifying the binary layout of file formats. Symbols in the regular expression refer to *structures* which define the building blocks of a file format. Each structure, in turn has a number of field declarations, with constraints on length or contents of the field.

There are 3 kinds of semantic entities in Derric: the file format, structures, and fields. Inside the regular expression, symbolic names refer to structures. Structures themselves refer to other structures to express inheritance. Finally, field constraints may refer to fields defined in other structures or defined locally in the enclosing structure.

[7] https://github.com/cwi-swat/textual-model-diff

In an earlier study, the authors of [17] investigated whether Derric could accommodate practical evolution scenarios on Derric programs. This has resulted in a public Github repository, containing the detailed history of three file format descriptions, for GIF, PNG and JPEG[8].

For each description, we have applied TMDIFF on subsequent revisions, and compared the resulting edit scripts to the ordinary textual diffs produce by the Git version control system[9]. The results are shown in Table 1. The first three columns identify the file and the two consecutive revisions (Git hashes) that have been compared. Column 4, 5 indicate the number of lines added and removed, as computed by the standard diff tool used by Git. To approximate the relative size of the changes, column 6 shows the number of line additions and removals per line of code in the source revision. The following eight columns then show how often each of the edit operations occurred in the delta computed by TMDIFF. The results are summarized in the next three columns, showing the total number of operations, the percentage indicating the number of operations per original AST node, and the number of nodes literally built by the delta. The last column contains the log message to provide an intuition of the intent of the revision.

Table 1 shows that some operations actually were never computed by TMDIFF. For instance, there are no Delete operations. This can be explained from the fact that, indeed, all revisions involve adding elements to the file descriptions; nothing is actually ever deleted.

The operations SetPrim and SetRef did not occur either. The reason is that there are no revisions at that level of granularity. Most changes are additions of structures and/or fields, or changes to the sequence constraints of a file format. In both cases, references and primitives end up as part of InsertTree operations. An example is shown in Fig. 9. The left and right columns show fragments of two versions of the GIF file format. The only change is and additional optional element at the end of the **sequence** section. The delta computed by TMDIFF is shown at the bottom of the figure. It consists of a single InsertTree operation. Within the inserted tree, one finds actual references to the structures CommentExtension and DataBlock.

The ratios of changes per total units of change (i.e. lines resp. AST nodes) show that TMDIFF deltas are consistently smaller that the ordinary textual deltas. It is also not the case that a single operation InsertTree operation replaces large parts of the model in one go. The before-last column shows that the number of nodes literally contained in a delta is reasonable. The largest number is 65 (fourth from below). As as comparison, the average number of nodes across all revisions in Table 1 is 432.

Figure 10 shows a typical delta computed by TMDIFF on a Derric description. It involves adding a new structure (`COMASC`) and its two fields (`length` and `data`). They are initialized in three InsertTree operations. The last three operations wire the newly created elements into the existing model.

[8] `https://github.com/jvdb/derric-eval`

[9] The actual command:
`git diff --patience --ignore-blank-lines --ignore-all-space` R_1 R_2 *path*.

File	R_1	R_2	+lines	−lines	$\{+,-\}$/LOC (%)	#Create	#Delete	#InsertTree	#InsertRef	#Remove	#SetPrim	#SetRef	#SetTree	Total	*#edits*/*#nodes* (%)	*#nodes* $\in \Delta$	Log message
gif.derric	`fc43456`	`2c28d2a`	2	2	2.8	1	0	0	1	2	0	0	1	5	1.2	8	Removed required value range on GraphicControlExtension.DisposalMethod.
	`2c28d2a`	`a3cb744`	2	2	2.8	0	0	1	0	1	0	0	0	2	0.5	12	Added optional GraphicControlExtension to initial CommentExtension subsequence.
	`a3cb744`	`7cd6500`	5	4	6.4	0	0	2	0	1	0	0	0	3	0.7	10	GraphicControlExtension is now optional in the main sequence.
	`7cd6500`	`cd76b13`	1	4	3.5	0	0	1	0	7	0	0	0	8	1.9	8	Removed last three fields from ApplicationExtension.
	`cd76b13`	`46379ec`	2	2	2.9	0	0	1	0	1	0	0	0	2	0.5	13	Added optional GraphicControlExtension to final CommentExtension subsequence.
	`46379ec`	`d09ac40`	2	2	2.9	0	0	1	0	1	0	0	0	2	0.5	1	Trailer is now optional.
	`d09ac40`	`9b3f919`	2	2	2.9	0	0	1	0	1	0	0	0	2	0.5	1	ZeroBlock is now optional in the main sequence.
	`9b3f919`	`872cd67`	2	1	2.2	0	0	1	0	0	0	0	0	1	0.2	2	Added optional CommentExtension subsequence with a single DataBlock and no ZeroBlock to main sequence.
png.derric	`d71a7c4`	`3922516`	22	2	17.1	10	0	1	5	1	0	0	10	27	7.8	32	Added private Macromedia (Adobe) Fireworks chunks prVW, mkBF, mkTS, mkBS and mkBT.
	`3922516`	`f97370b`	6	2	5.	2	0	1	1	1	0	0	2	7	1.8	8	Added vpAg structure.
	`f97370b`	`3780274`	6	2	4.9	2	0	1	1	1	0	0	2	7	1.8	8	Added oFFs structure.
	`3780274`	`cc7f2f3`	6	2	4.8	2	0	1	1	1	0	0	2	7	1.7	8	Added tpNG structure.
	`cc7f2f3`	`7c32673`	6	1	4.1	2	0	1	1	0	0	0	2	6	1.4	7	Added bBPn structure.
	`7c32673`	`454152a`	10	2	6.8	4	0	1	2	1	0	0	4	12	2.8	14	Added cmOD and cpIp structures.
	`454152a`	`bdbf985`	6	2	4.3	2	0	1	1	1	0	0	2	7	1.6	8	Added meTa structure.
	`bdbf985`	`3caa428`	6	2	4.2	2	0	1	1	1	0	0	2	7	1.5	8	Added eXIF structure.
	`3caa428`	`6b0cca9`	2	2	2.1	0	0	1	0	1	0	0	0	2	0.4	2	Modified sequence to allow the oFFs structure to occur after the bKGD structure.
	`6b0cca9`	`ec33a53`	2	2	2.1	0	0	1	0	1	0	0	0	2	0.4	2	Modified sequence to allow the bKGD to occur before the PLTE structure.
	`ec33a53`	`fddce35`	2	2	2.1	0	0	1	0	1	0	0	0	2	0.4	1	IEND is now optional.
	`fddce35`	`20b63f0`	6	2	4.1	2	0	1	1	1	0	0	2	7	1.5	8	Added gIFg structure.
	`20b63f0`	`b8cd1d9`	6	2	4.1	2	0	1	1	1	0	0	2	7	1.5	8	Added tpNg structure.
	`b8cd1d9`	`f096d6c`	6	2	4.	2	0	1	1	1	0	0	2	7	1.4	8	Added cmPP structure.
	`f096d6c`	`cff3430`	10	2	5.9	4	0	1	2	1	0	0	4	12	2.4	14	Added acTL and fcTL structures.
	`cff3430`	`a691cde`	2	2	1.9	0	0	1	0	1	0	0	0	2	0.4	2	Modified sequence to allow the vpAg to occur before the PLTE structure.
	`a691cde`	`bdc85e9`	6	2	3.8	2	0	1	1	1	0	0	2	7	1.4	8	Added pRVW structure.
	`bdc85e9`	`399fb54`	2	2	1.8	0	0	1	0	1	0	0	0	2	0.4	2	Modified sequence to allow the cmOD, cpIp and meTa structures to occur before the IDAT structure.
jpeg.derric	`590a396`	`c1b3578`	7	2	10.3	3	0	2	1	2	0	0	3	11	3.8	28	Added APP0Picasa.
	`c1b3578`	`6ebbad4`	2	2	4.3	0	0	2	0	2	0	0	0	4	1.3	18	Modified sequence to allow APP0JFXX to appear as first APP structure.
	`6ebbad4`	`ef0329b`	10	2	13.	5	0	2	1	2	0	0	5	15	4.9	37	Added APP14Adobe.
	`ef0329b`	`d679520`	10	2	12.	5	0	2	1	2	0	0	5	15	4.5	37	Added APP13Photoshop.
	`d679520`	`fce26b3`	2	2	3.7	0	0	2	0	2	0	0	0	4	1.1	19	The APP-only sequence is now optional.
	`fce26b3`	`bbe0bf1`	4	2	5.6	0	0	2	1	1	0	0	0	4	1.1	3	EOI is no longer required, but SOS is now required.
	`bbe0bf1`	`13f1e56`	4	3	6.4	2	0	3	1	3	0	0	2	11	2.9	23	Added SOF1 structure.
	`13f1e56`	`6a8b0d7`	14	8	19.8	5	0	6	4	11	0	0	5	31	7.9	65	Added 0xFF padding.
	`6a8b0d7`	`acfab2d`	5	3	6.8	2	0	3	1	3	0	0	2	11	2.3	59	Added SOF3 structure.
	`acfab2d`	`712e583`	7	1	6.7	2	0	3	1	1	0	0	2	9	1.8	47	Added COMElanGmk variant of COM.
	`712e583`	`afb17f7`	8	1	7.2	3	0	2	1	0	0	0	3	9	1.6	15	Added COMASC variant of COM.

Table 1: Applying TMDIFF to revisions of Derric fileformat specifications.

```
format gif d0
sequence
  (Header87a Header89a)
  LogicalScreenDesc
  GraphicControlExtension?
  (
    [TableBasedImage CompressedDataBlock*]
    [PlainTextExtension DataBlock*]
    [ApplicationExtension DataBlock*]
    [CommentExtension DataBlock*]
  )
  ZeroBlock?
  (
    [GraphicControlExtension?
      TableBasedImage CompressedDataBlock*
      ZeroBlock]
    [GraphicControlExtension?
      PlainTextExtension DataBlock* ZeroBlock]
    [ApplicationExtension DataBlock* ZeroBlock]
    [GraphicControlExtension? CommentExtension
      DataBlock* ZeroBlock]
  )*
  Trailer?
...
CommentExtension d1  = ...
DataBlock d2  = ...
```

```
format gif
sequence
  (Header87a Header89a)
  LogicalScreenDesc
  GraphicControlExtension?
  (
    [TableBasedImage CompressedDataBlock*]
    [PlainTextExtension DataBlock*]
    [ApplicationExtension DataBlock*]
    [CommentExtension DataBlock*]
  )
  ZeroBlock?
  (
    [GraphicControlExtension?
      TableBasedImage CompressedDataBlock*
      ZeroBlock]
    [GraphicControlExtension?
      PlainTextExtension DataBlock* ZeroBlock]
    [ApplicationExtension DataBlock* ZeroBlock]
    [GraphicControlExtension? CommentExtension
      DataBlock* ZeroBlock]
  )*
  [CommentExtension DataBlock]?
  Trailer?
...
```

```
d0.sequence[6] = Optional(Seq([d1, d2]))
```

Fig. 9: A minimal change to the sequence part of a Derric description of GIF. A single line is added on right (underlined). At the bottom the edit script computed by TMDIFF (between `9b3f919` and `872cd67`)

5 Discussion and Related Work

The case-study of the previous section shows that TMDIFF computes reasonable deltas on realistic evolution scenarios on DSL programs. In this section we discuss a number of limitations of TMDIFF and directions for further research.

The matching of entities uses textual deltas computed by `diff` as a guiding heuristic. In rare cases this affects the quality of the matching. For instance, `diff` works at the granularity of a line of code. As a result, *any* change on a line defining a semantic entity will incur the entity to be marked as added. The addition of a single comment may trigger this incorrect behavior. Furthermore, if a single line of code defined multiple entities, a single addition or removal will trigger the addition of all other entities. Nevertheless, we expect entities to be defined on a single line most of the time.

If not, the matching process can be made immune to such issues by first pretty-printing a textual model (without comments) before performing the textual comparison. The pretty-printer can then ensure that every definition is on its own line. Note, that simply projecting out all definition names and performing longest common subsequence (LCS) on the result sequences abstracts from a lot of textual context that is typically used by `diff`-like tools. In fact, this was our first approach to matching. The resulting matching, however, contained significantly more false positives.

Another factor influencing the precision of the matchings is the dependence on the textual order of occurrence of names. As a result, when entities are moved around

```
format d4 jpeg
sequence
  SOI
  PADDING*
  COMASC?
  ...

PADDING d5 = ...
COM d3 = ...

COMASC d2 = COM {
 length d1 : lengthOf(data) size 2;
 data d0 : "Created by AccuSoft Corp.", 0;
}
```

```
create Field d0
create Field d1
create Term d2
d0 = Field("data",[Exps([
  Str("Created␣by␣AccuSoft␣Corp."),
  Num(0)])])
d1 = Field("length",[Exps([d0]),
  Qualifier(Size(Num(2)))])
d2 = Struct("COMASC",d3,[d1,d0])
d4.sequence[1] = Iter(d5)
d4.sequence[2] = Optional(d2)
d4.structs[21] = d2
```

Fig. 10: Fragment of revision `afb17f7` of `jpeg.derric` (left, added lines are underlined), and the relevant part of the TMDIFF delta from revision `712e583` to `afb17f7` (right).

without any further change, TMDIFF will not detect it. We have experimented with a simple move detection algorithm to mitigate this problem, however, this turned out to be too computationally expensive. Fortunately, edit distance problems with moves are well-researched, see, e.g., [15]. A related problem is that TMDIFF will always see renames as an addition and removal of an entity. Further research is needed if renames of entities can be detected, for instance by matching up additions and removals of entities, where the deleted node and the added node are the same, modulo the renaming.

Much work has been done in the research area of model comparison that relates to TMDIFF. We refer to a survey of model comparison approaches and applications by Stephan and Cordy for an overview [14]. In the area of model comparison, *calculation* refers to identifying similarities and differences between models, *representation* refers to the encoding form of the similarities and differences, and *visualization* refers to presenting changes to the user [9, 14]. Here we focus on the calculation aspect.

Calculation involves matching entities between model versions. Strategies for matching model elements include matching by 1) *static identity*, relying on persistent global unique entity identifiers; 2) *structural similarity*, comparing entity features; 3) *signature*, using user defined comparison functions; 4) *language specific algorithms* that use domain specific knowledge [14]. With respect to this list, our approach represents a new point in the design space: matching by textual alignment of names.

The differencing algorithm underlying TMDIFF is directly inspired by Alanen and Porres' seminal work [1]. The identification map M between model elements is explicitly mentioned, but the main algorithm assumes that model element identities are stable. Additionally, TMDIFF supports elements without identity. In that case, TMDIFF performs a structural diff on the containment hierarchy (see, e.g., [20]).

TMDIFF's differencing strategy resembles the model merging technique used Ensō [18]. The Ensō "merge" operator also traverses a spanning tree of two models in parallel and matches up object with the same identity. In that case, however, the objects are identified using primary keys, relative to a collection (e.g., a set). This means that matching only happens between model elements at the same syntactic level of the spanning tree of an Ensō model. As a result, it cannot deal with "scope travel" as in Fig. 2c, where the `locked` state moved from the global state to the `locking` scope. On the other hand, the matching is more precise, since it is not dependent on the heuristics of textual alignment.

Epsilon is a family of languages and tools for model transformation, model migration, refactoring and comparison [10]. It integrates HUTN [13], the OMG's Human Usable Text Notation, to serialize models as text. As result, which elements define semantic identities is known for each textual serialization. In other words, unlike in our setting, HUTN provides a fixed concrete syntax with fixed scoping rules. TMDIFF allows languages to have custom syntax, and custom binding semantics.

6 Conclusion

Accurately differencing models is important for managing and supporting the evolution of models. Representing models as text, however, poses a challenge for model differencing algorithms, because the identity of model elements is not stable across revisions.

In this paper we have shown how this challenge could be addressed by constructing the mapping between model elements using origin tracking and traditional textual differencing. Origin tracking traces the identity of an element back to the symbolic name that defines it in the textual source of a model. Using textual differencing these names can be aligned between versions of a model. Combining the origin relation and the alignment of names is sufficient to identify the model elements themselves. It then becomes possible to apply standard model differencing algorithms.

Based on these techniques, we have presented TMDIFF, a fully language parametric approach to textual model differencing. A prototype of TMDIFF has been implemented in the RASCAL meta programming language [8]. The prototype was used to illustrate the feasibility of TMDIFF by reconstructing the version history of existing textual models. The models in question are file format descriptions in an independently developed DSL in the domain of in digital forensics [16].

Although the work presented in this paper shows promise, important directions for further research remain. First of all, it is unclear if the deltas produced by TMDIFF are on average smaller than the deltas produced by, for instance, EMFCompare [3], for languages which have scoping aligned with the containment hierarchy. Further evaluation should also include benchmarking the size and speed of differencing against a broader set of practical examples.

References

1. Alanen, M., Porres, I.: Difference and Union of Models. In: UML. (2003) 2–17
2. Brun, C., Pierantonio, A.: Model Differences in the Eclipse Modeling Framework. UPGRADE, The European Journal for the Informatics Professional **9**(2) (2008) 29–34
3. Eclipse Foundation: EMF Compare Project. `https://www.eclipse.org/emf/compare/`
4. Erdweg, S., van der Storm, T., Völter, M., et al.: The State of the Art in Language Workbenches. In: SLE. Volume 8225 of LNCS., Springer (2013) 197–217

5. Eysholdt, M., Behrens, H.: Xtext: Implement Your Language Faster Than the Quick and Dirty Way. In: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion. OOPSLA '10, New York, NY, USA, ACM (2010) 307–309
6. Goldschmidt, T., Becker, S., Uhl, A.: Classification of concrete textual syntax mapping approaches. In: Proceedings of the European Conference on Model Driven Architecture—Foundations and Applications (ECMDA-FA). Volume 5095 of LNCS. (2008) 169–184
7. Inostroza, P., van der Storm, T., Erdweg, S.: Tracing program transformations with string origins. In Di Ruscio, D., Varró, D., eds.: Theory and Practice of Model Transformations. Volume 8568 of LNCS. Springer (2014) 154–169
8. Klint, P., van der Storm, T., Vinju, J.: Rascal: A Domain-Specific Language for Source Code Analysis and Manipulation. In: SCAM. (2009) 168–177
9. Kolovos, D.S., Di Ruscio, D., Pierantonio, A., Paige, R.F.: Different Models for Model Matching: An Analysis of Approaches to Support Model Differencing. In: ICSE Workshop on Comparison and Versioning of Software Models (CVSM'09), IEEE (2009) 1–6
10. Kolovos, D.S., Paige, R.F., Polack, F.A.: The Epsilon Transformation Language. In: Theory and practice of model transformations. Springer (2008) 46–60
11. Miller, W., Myers, E.W.: A File Comparison Program. Softw. Pract. Exper. **15**(11) (1985) 1025–1040
12. Myers, E.W.: An $O(ND)$ Difference Algorithm and its Variations. Algorithmica **1**(1-4) (1986) 251–266
13. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.: Constructing Models with the Human-Usable Textual Notation. In Czarnecki, K., Ober, I., Bruel, J.M., Uhl, A., Völter, M., eds.: Model Driven Engineering Languages and Systems. Volume 5301 of LNCS. Springer Berlin Heidelberg (2008) 249–263
14. Stephan, M., Cordy, J.R.: A Survey of Model Comparison Approaches and Applications. In: MODELSWARD. (2013) 265–277
15. Tichy, W.F.: The String-to-string Correction Problem with Block Moves. ACM Trans. Comput. Syst. **2**(4) (1984) 309–321
16. van den Bos, J., van der Storm, T.: Bringing Domain-Specific Languages to Digital Forensics. In: ICSE'11, ACM (2011) Software Engineering in Practice.
17. van den Bos, J., van der Storm, T.: A Case Study in Evidence-based DSL Evolution. In: ECMFA'13, Springer (2013) 207–219
18. van der Storm, T., Cook, W.R., Loh, A.: The Design and Implementation of Object Grammars. Science of Computer Programming **96, Part 4**(0) (2014) 460–487 Selected Papers from the Fifth International Conference on Software Language Engineering (SLE 2012).
19. van Deursen, A., Klint, P., Tip, F.: Origin Tracking. Symbolic Computation **15** (1993) 523–545
20. Yang, W.: Identifying Syntactic Differences Between Two Programs. Softw. Pract. Exper. **21**(7) (1991) 739–755