Model Differencing for Textual DSLs

Riemer van Rozen
Amsterdam University of Applied Sciences (HvA)
Amsterdam, The Netherlands
Email: r.a.van.rozen@hva.nl

Tijs van der Storm
Centrum Wiskunde & Informatica (CWI)
Amsterdam, The Netherlands
Email: storm@cwi.nl

***Abstract*—The syntactic and semantic comparison of models is important for understanding and supporting their evolution. In this paper we present TMDIFF, a technique for semantically comparing models that are represented as text. TMDIFF incorporates the referential structure of a language, which is determined by symbolic names and language-specific scoping rules. Furthermore, it employs a novel technique for matching entities existing in source and target versions of a model, and finds entities that are added or removed. As a result, TMDIFF is fully language parametric, and brings the benefits of model differencing to textual languages.**

I. Introduction

Model differencing is a well-researched topic in the context of Model-Driven Engineering (MDE). For instance, the seminal paper by Alanen and Porres [1] introduced a generic algorithm to compute the difference and union between two models. In this paper we introduce Textual Model Diff (TMDIFF): an adaption of the Alanen and Porres algorithm for models represented as textual source code. Textual representation is common in the area of domain-specific languages (DSLs). We expect that TMDIFF will pose new opportunities to better understand and support the evolution of DSL programs in a similar way that the MDE process is supported by numerous tools for comparing, merging, and migrating models.

Applying model-based differencing techniques to textual models is non-trivial for two reasons. First, the referential structure of a textual model is encoded using symbolic names and language specific scoping rules. Second, textual languages are dependent on parsing for obtaining a structured representation. As a result, model elements do not have a stable identity across versions of a model.

TMDIFF addresses these problems as follows. First, TMDIFF is parameterized in the name binding semantics of the modeling language using a generic, relation-based representation of references. Second, the identities of entities across revisions of a model are recovered by aligning their defining name occurrences using stock `diff` algorithms (e.g.,[4]).

Below we present a motivating example based on textual state machine models. Then we present an overview of TMDIFF. We conclude with a discussion on limitations and future work.

II. Motivating Example

Figure 1 shows three versions of a textual model in a simple language for state machines. 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. Figure 1a displays a state machine for controlling doors ($Doors_1$). The state machine is extended with a `locked` state in $Doors_2$ (Fig. 1b). The third version, $Doors_3$ (Fig. 1c), 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.

```
machine doors d1
 state closed d2
  open => opened u1

 state opened d3
  close => closed u2

end
```

(a) $Doors_1$

```
machine doors d4
 state closed d5
  open => opened u3
  lock => locked u4

 state opened d6
  close => closed u5

 state locked d7
  unlock => closed u6

end
```

(b) $Doors_2$

```
machine doors d8
 state closed d9
  open => opened u7
  lock =>
    locking.locked u8

 state opened d10
  close => closed u9

 locking d11  {
  state locked d12
    unlock => closed u10
 }
end
```

(c) $Doors_3$

Fig. 1. Three versions of a simple state machine model. Definitions and uses of states are labeled with d_i and u_j respectively.

In each of the state machine models, the constructs that *define* entities are annotated with unique labels `dn`. For instance, in $Doors_1$, the machine itself is labeled `d1`, and both states `closed` and `opened` are labeled `d2` and `d3` respectively. Similarly, *uses* of states in transitions are labeled with labels `un`. For instance, the target state `opened` of the transition in `closed` is labeled `u1`.

To the human reader it is intuitively clear that states `closed` and `opened` are stable across revisions: only the relations between these states and other states are changed through the addition or change of transitions and addition of new states (`locked`). Textual or structural difference algorithms, however, are oblivious to the semantic identity of states and will generate spurious differences as a result.

TMDIFF does take into account constructs that represent semantic entities. It reports differences as imperative edit scripts in terms of a metamodel that is implicitly derived from the grammar of the language and its name binding semantics. For instance the difference between $Doors_1$ and $Doors_2$ is reported as:

```
create State d7                          //create State def
d7 = State("locked",[Trans("unlock",d2)]) //init new State
d2.out[1] = Trans("lock", d7)            //store 2nd Trans
d1.states[2] = d7                        //store new State
```

A new state `d7` (`locked`) is created and initialized to contain a single transition to the (existing) state `d2`. Then the `closed` state gets a new transition to the state that was just created (`d7`). Finally, state `d7` is added to the list of states of the state machine `d1`.

To illustrate the fact that TMDIFF can deal with scoping constructs, consider the difference between $Doors_2$ and $Doors_3$. Informally, the only thing that is changed is that the `locked` state is placed in a scope called `locking`. As a result, the reference to the locked state `u3` in $Doors_2$ needs to be updated to use the qualified name `locking.locked`. However, semantically the transition structure between states does not change. The edit script produced by TMDIFF accurately reflects this description:

```
create Group d11                         //create Group def
d11 = Group("locking",[d7])              //initialize new Group
remove d4.states[2]                      //remove 3rd State
d4.states[2] = d11                       //store new Group
```

The script first creates the `Group` construct `d11` and then initializes its name to `locking` and its owned states to contain a pointer to `locked`. Next, that state is removed from the list of states of the machine. Finally, the newly created group becomes the third element in this list. Everything else stays the same.

III. Overview of TMDIFF

TMDIFF is based on two relations: the reference relation between entities in a single model, and a matching relation between entities in different versions of the same model. We briefly describe each in turn.

a) Name Analysis: The user-specified name analysis should produce *reference graphs* in terms of definition and reference labels. A reference graph is triple $G = \langle D, U, R \rangle$, where D and U are sets of labels identifying definitions and uses respectively, and $R \subseteq (U \cup D) \times D$ is a binary relation representing references.

Figure 2 shows the abstract syntax tree (AST) and reference graph of $Doors_1$. The dashed arrows represent reference tuples in R. For instance, the `Ref` node ("opened") is labeled `u1` and refers to `d3`, the label of a `Name` node ("opened").

The reference graph provides two important pieces of information: namely, the AST nodes representing definitions of entities, and nodes that are references to such entities.

b) Matching Entities: The matching process takes the textual source of both models, their ASTs and their reference graphs as input. It first creates *entity projections*, P_1 and P_2 which are sequences of tuples $\langle x, c, l, d \rangle$, where x is the symbolic name of the entity, c its semantic category (e.g. State, Machine, etc.), l the textual line it occurs on and d its definition label (e.g., d_1). For instance, The entity projections for $Doors_1$ and $Doors_2$ are as follows:

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

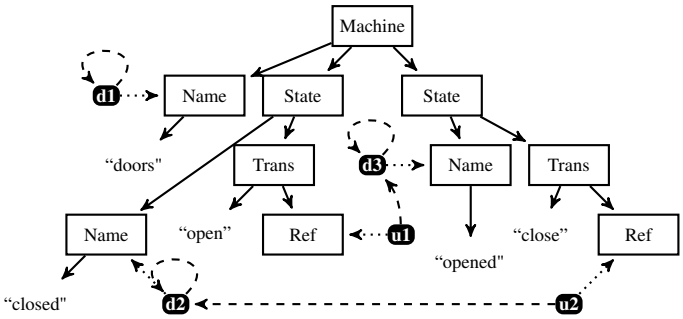

Fig. 2. AST with references of $Doors_1$. Solid arrows indicate containment in the AST. Dotted lines associate unique labels to AST nodes, and dashed lines are references.

The line numbers in the entity projections provide the key to using a traditional textual `diff` to determine whether an entity has been added or not. Recall that `diff` produces a patch describing which lines we added or removed. In the context of $Doors_1$ and $Doors_2$, for instance, the first three entries in both P_1 and P_2 all have line numbers that are not changed by the diff. Therefore the labels d_1, d_2, d_3 and d_4, d_5, d_6 are pairwise matched. Entity d_7 however was defined on line 9, and this is one of the lines marked as added by the textual diff. As result, d_7 is considered to represent a newly created entity. Recovering deleted entities works the other way round.

The reference graph provides the information on which nodes are actually semantic entities, and how entities refer to each other. Entity matching determines which entities exist in both revisions of a textual model. Together the reference graph and the entity matching represent the necessary information for applying existing model differencing algorithms such as [1].

IV. Discussion and Outlook

We have implemented a prototype of TMDIFF in Rascal, a meta programming language and environment for source code analysis and transformation [2]. As an initial experiment, we were able to reconstruct the complete history of file description models used in a DSL for digital forensics [5].

Future work is aimed at assessing how our `diff`-based matching strategy compares to existing approaches [3]. In particular, our strategy is not resilient against moving around of definitions, since traditional `diff` will not detect them as such. We are also investigating how our generic textual model differences can be used for reconciling co-evolving artifacts and migrating run-time states of textual DSL programs.

References

[1] M. Alanen and I. Porres. Difference and Union of Models. In *UML*, pages 2–17, 2003.

[2] P. Klint, T. van der Storm, and J. J. Vinju. RASCAL: A domain specific language for source code analysis and manipulation. In *SCAM*, pages 168–177. IEEE, 2009.

[3] D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige. Different Models for Model Matching: An Analysis of Approaches to Support Model Differencing. CVSM '09, pages 1–6, Washington, DC, USA, 2009. IEEE Computer Society.

[4] W. Miller and E. W. Myers. A File Comparison Program. *Softw. Pract. Exper.*, 15(11):1025–1040, 1985.

[5] J. van den Bos and T. van der Storm. A Case Study in Evidence-based DSL Evolution. In *ECMFA'13*, pages 207–219. Springer, 2013.