Softw Syst Model
DOI 10.1007/s10270-017-0608-7

SPECIAL SECTION PAPER

Toward live domain-specific languages

From text differencing to adapting models at run time

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

Received: 27 June 2016 / Revised: 26 May 2017 / Accepted: 20 June 2017
© Springer-Verlag GmbH Germany 2017

Abstract Live programming is a style of development characterized by incremental change and immediate feedback. Instead of long edit-compile cycles, developers modify a running program by changing its source code, receiving immediate feedback as it instantly adapts in response. In this paper, we propose an approach to bridge the gap between running programs and textual domain-specific languages (DSLs). The first step of our approach consists of applying a novel model differencing algorithm, TMDIFF, to the textual DSL code. By leveraging ordinary text differencing and origin tracking, TMDIFF produces deltas defined in terms of the metamodel of a language. In the second step of our approach, the model deltas are applied at run time to update a running system, without having to restart it. Since the model deltas are derived from the static source code of the program, they are unaware of any run-time state maintained during model execution. We therefore propose a generic, dynamic patch architecture, RMPATCH, which can be customized to cater for domain-specific state migration. We illustrate RMPATCH in a case study of a live programming environment for a simple DSL implemented in RASCAL for simultaneously defining and executing state machines.

Keywords Live programming · Domain-specific languages · Text differencing · Model patching · Adapting models · Models at run time

Communicated by Prof. Alfonso Pierantonio, Jasmin Blanchette, Francis Bordeleau, Nikolai Kosmatov, Prof. Gabriele Taentzer, Prof. Manuel Wimmer.

✉ Riemer van Rozen
R.A.van.Rozen@hva.nl

Tijs van der Storm
T.van.der.Storm@cwi.nl

1 Amsterdam University of Applied Sciences, PO Box 1025, 1000 BA Amsterdam, The Netherlands

2 Centrum Wiskunde & Informatica, PO Box 94079, 1090 GB Amsterdam, The Netherlands

3 University of Groningen, Johann Bernoulli Institute, Nijenborgh 9, 9747 AG Groningen, The Netherlands

1 Introduction

The "gulf of evaluation" represents the cognitive gap between an action performed by a user and the feedback provided to her about the effect of that action [23]. Live programming aims to bridge the gulf of evaluation by shortening the feedback loop between editing a program's textual source code and observing its behavior. In a live programming environment, the running program is updated instantly after every change in the code [34]. As a result, developers immediately see the behavioral effects of their actions and learn predicting how the program adapts to targeted improvements to the code. In this paper, we are concerned with providing generic, reusable frameworks for developing "live DSLs", languages whose users enjoy the immediate feedback of live execution. We consider such techniques to be first steps toward providing automated support for live languages in language workbenches [8].

In particular, we propose two reusable components, TMDIFF and RMPATCH to ease the development of textual live DSLs, based on a foundation of metamodeling and model interpretation. TMDIFF is used to obtain model-based deltas from textual source code of a DSL. These deltas are then applied at run time by RMPATCH to migrate the execution of the DSL program [38]. This enables the users of a DSL to modify the source and immediately see the effect.

Published online: 14 August 2017

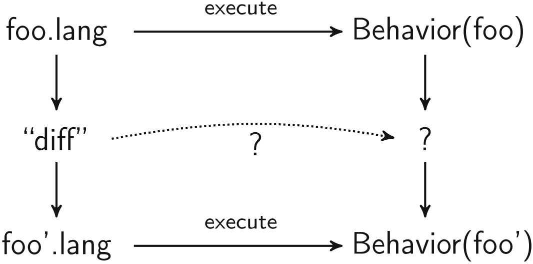


Fig. 1 How to get from a textual difference between source code versions to a runtime difference in behavior?

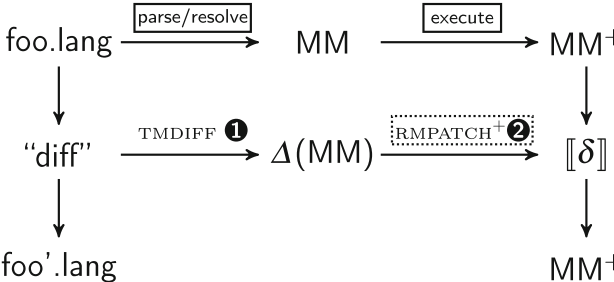


Fig. 2 Applying TMDIFF to obtain model-based deltas and RMPATCH to migrate models at runtime

The first component of our approach is the TMDIFF algorithm [43]. TMDIFF employs textual differencing and origin tracking to derive model-based deltas from changes in textual source code. A textual difference is translated to a difference on the abstract syntax of the DSL, as specified by a metamodel. As a result, standard model differencing algorithms (e.g., [1]) can be applied in the context of textual languages.

The second component, RMPATCH, is used to dynamically adapt model execution to changes in the source code. This is achieved by "patching" the execution using the deltas produced by TMDIFF. We call differences applied to running programs *executable deltas*. To apply executable deltas, we require that a language is implemented as a model interpreter [30]. In particular, we require that every class defined in a language's metamodel has an implementation counterpart in some programming language (we use Java). The RMPATCH architecture supports applying an executable delta on the instances of those classes while the model is interpreted. To support run-time state, we allow the run-time classes to extend the classes of the metamodel with additional attributes and relations. Since the deltas produced by TMDIFF are unaware of those attributes and relations, the RMPATCH engine is designed to be open for extension to cater for migrating such domain-specific run-time state. RMPATCH has been applied in the development of a prototype live programming environment for a simple state machine DSL. A state machine definition can be changed while it is running, and the runtime execution will adapt instantly.

The key contribution of this paper is the combination of textual model differencing and run-time model patching for adapting models at run time with "live" textual DSLs, and to this end:

- We reiterate how textual differencing can be used to match model elements based on origin tracking information and provide a detailed description of TMDIFF, including a prototype implementation (Sect. 3).
- We present a generic architecture for run-time patching of interpreted models (Sect. 4).
- We illustrate the framework using a live DSL environment for a simple state machine language (Sect. 5).

This article is an extended version of our previous work "Origin Tracking + Text Differencing = Textual Model Differencing," published in *Theory and Practice of Model Transformations*, ICMT, 2015 [43]. In particular, the present paper extends the work with the patch architecture (RMPATCH), as well as the live state machine case study. For the evaluation of TMDIFF itself, we refer to the original paper [43].

2 From text differencing to live models at run time

We motivate our work by taking the perspective of developers who use textual DSLs to iteratively modify and improve programs. Figure 1 gives an overview of the challenge of bridging the gap between a developer's textual model edits and the associated program behavior that the developer needs to quickly observe, understand and improve.

A developer writes a program (foo) in some language (lang), which can be executed to obtain its behavior. The developer then evolves the program to a new version (foo') by updating its source, yielding a textual difference. In a traditional setting, the effect of the change can only be observed by re-executing the program. However, this involves compiling and executing the program from scratch. This can be a time-consuming distraction, losing all dynamic context observed while running foo. In particular, all run-time state accumulated during the execution of program version foo is lost when its next version foo' is executed (again). We aim to make this experience more fluid and live by obtaining a "run-time diff" from the textual "diff" between successive program versions (foo and foo') and then migrating its execution (from Behavior(foo) to Behavior(foo')) at run time.

Figure 2 shows an overview of our solution to this problem. The foo program is mapped to an instance of a metamodel (MM), through parsing and name resolution. Parsing constructs an initial containment hierarchy of the program in the form of an abstract syntax tree (AST). Name resolution, on the other hand, creates cross-references in the model based on the (domain-specific) referencing and scoping rules of the language, yielding an abstract syntax graph (ASG). The model is then executed by an interpreter, which creates a

run-time model corresponding to foo. This run-time model is an instance of an enhanced metamodel (MM^+), representing run-time state as additional attributes and relations. We require that MM^+ is an extension of MM.

Whenever the developer evolves the program's source, the textual difference between foo and foo' is now mapped to a model-based delta over the metamodel MM using TMDIFF. Such a delta consists of an edit script which changes the model of foo to a model representing foo'. That delta is then applied as an executable delta to the executing run-time model of foo by RMPATCH. Because the executing model has additional run-time state that could become invalid, RMPATCH needs to be augmented with language-specific migrations. The generic part of RMPATCH will only migrate the parts defined by MM; the domain-specific customization defines what to do with the extensions defined by MM^+. At specific points during execution, the interpreter will swap out the old version of the model and start executing the new one, without having to restart, and without losing state.

Note that the parts in boxes are the components that are language specific. This includes parsing and name resolution, which often need to be defined anyway, and a model-based interpreter. TMDIFF is completely language parametric and thus can be reused for multiple live DSLs. RMPATCH is partially generic: it is generically defined for deltas produced by TMDIFF, but needs to be extended for dealing with the run-time state extensions defined by MM^+.

The rest of the paper is structured as follows. Next in Sect. 3, we describe how TMDIFF works. In Sect. 4, we show how the deltas produced by TMDIFF are applied at run time using the generic patch architecture of RMPATCH. The customization of this architecture to support run-time state migration is described as part of our case study based on state machines in Sect. 5. We show how this enables a live programming environment for state machines using a prototype interpreter. We conclude the paper with a discussion of related work and an outline for further research.

3 TMDiff: textual model diff

3.1 Overview

TMDIFF is a novel differencing algorithm that leverages ordinary text differencing and origin tracking to derive model-based deltas from textual source code. Traditional 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 is 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 [11], models are represented as textual source code, similar to DSLs and programming languages.

The actual model structure represented by an abstract syntax graph (ASG) is not first-class, but is derived from the text by a text-to-model mapping, which apart from parsing the text into an abstract syntax tree (AST) specifying a containment hierarchy also provides for reference resolution. After every change in 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 [9]) and then using standard model comparison tools (e.g., EMFCompare [3,6]). 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 and identity, irrespective of their location in the containment hierarchy of the model. Other approaches are to match elements in collections at the same position in the containment hierarchy, to use similarity-based heuristics or to construct a purpose-built algorithm.

Unfortunately, each of these approaches has its 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 terms of scoping. While similarity-based matching techniques can deal with scopes, these may also require fine-tuning the heuristic to obtain more accurate results for specific languages and uses.

TMDIFF is a language-parametric technique for model differencing of textual 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 [13,40]. 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

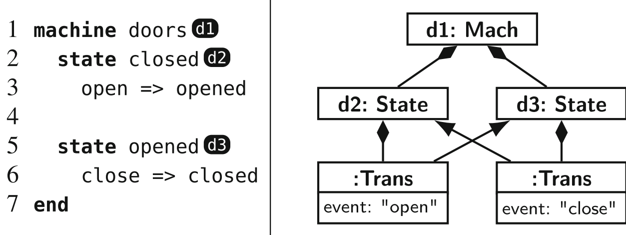


Fig. 3 $Doors_1$: a simple textual representation of a state machine and its model

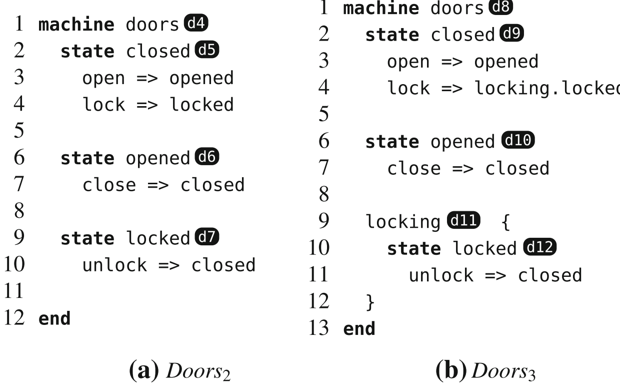


Fig. 4 Two new versions of the simple state machine model $Doors_1$

(e.g., [28]). 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.

Here we introduce textual model differencing using a simple motivating example that is used as a running example throughout the paper. Figure 3 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.

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.[1] Since identifiers can never overlap, labels are guaranteed to be unique, and the actual name corresponding to a 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. 3, 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., [45]), instead of semantic, model-based differencing [1].

Figure 4 shows two additional versions of the state machine of Fig. 3. First, the machine is extended with a `locked` state in $Doors_2$ (Fig. 4a). Second, $Doors_3$ (Fig. 4b) 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 Figs. 3 and 4, 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 result 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.

The source locations, therefore, cannot be used as (stable) identities 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, we identify which model elements have persisted after changing the source text.

This high-level approach is visualized in Fig. 5. 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

[1] 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.

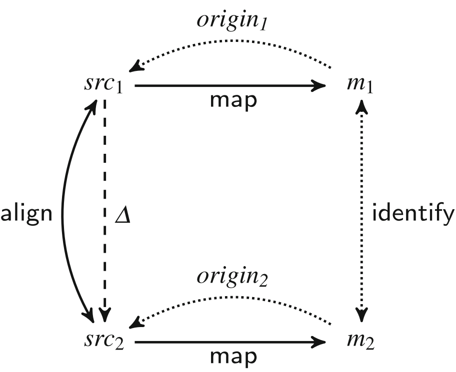


Fig. 5 Identifying model elements in m_1 and m_2 through origin tracking and alignment of textual names

```
--- 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. 6 Textual diff between $Doors_1$ and $Doors_2$, and $Doors_2$ and $Doors_3$. 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>`

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.

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 $Doors_3$, are shown in Fig. 6. As we can see, 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. 6 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 values to

```
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. 7 TMDIFF differences between $Doors_i$ and $Doors_{i+1}$ ($i \in \{1, 2\}$)

fields, and inserting or removing elements from collection-valued properties. Figure 7 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. 7a 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. 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. 7b. 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.

3.2 TMDiff in more detail

Top-level algorithm

Figure 8 shows the TMDIFF algorithm in high-level pseudocode. Input to the algorithm is the source texts of the

```
1  list[Operation] tmDiff(str src1, str src2, obj m1, obj m2) {
2    <A, D, M> = match(src1, src2, m1, m2)
3    Δ = [ new Create(da, da.class) | da ←A ]
4    M' = M + { <da, da> | da ←A }
5    Δ += [ new SetTree(da, build(da, M')) | da ←A ]
6    for (<d1, d2> ←M)
7      Δ += diffNodes(d1, d1, d2, [], M')
8    Δ += [ new Delete(dd) | dd ←D ]
9    return Δ
10 }
```

Fig. 8 TMDIFF

```
1   Matching match(str src1, str src2, obj m1, obj m2) {
2     P1 = project(m1)
3     P2 = project(m2)
4     <Ladd, Ldel> = split(diff(src1, src2))
5
6     i = 0, j = 0; A = {}, D = {}; I = {}
7     while (i < |P1| ∨ j < |P2|) {
8         if (i < |P1| ∧ P1[i].line ∈ Ldel)
9           D += {P1[i].object}; i += 1; continue
10        if (j < |P2| ∧ P2[j].line) ∈ Ladd)
11          A += {P2[j].object}; j += 1; continue
12        if (P1[i].object.class = P2[j].object.class)
13          I += {<P1[i].object, P2[j].object>}
14        else
15          D += {P1[i].object}; A += {P2[j].object}
16        i += 1; j += 1
17    }
18    return <A, D, I>;
19  }
```

Fig. 9 Matching model elements based on source text diffs

models (src_1, src_2), and the models themselves (m_1, m_2). The first step is to determine corresponding elements in m_1 and m_2 using the matching technique introduced above. We further describe the match function later in this section.

Based on the matching returned by match (line 2), TMDIFF first generates global Create operations for nodes that are in the A set (line 3). After these operations are created, the matching M is "completed" into M', by mapping every added object to itself (line 4). 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 (line 5). The function diffNodes then computes the difference between each pair of nodes originally identified in M (lines 6–7). 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 (line 8).

Matching

The match function uses the output computed by standard diff tools. In particular, we employ a diff variant called *Patience Diff*[2] which is known to often provide better results than the standard, LCS-based algorithm [31].

The matching algorithm is shown in Fig. 9. 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 (lines 1–2). 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 is its class (e.g., state, machine, etc.), l is the textual line it occurs on, and d is the object itself.

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

$$P_1 = \begin{array}{l} [\ \langle \texttt{doors}, \ Machine, \ 1, \ d_1\rangle, \\ \ \ \langle \texttt{closed}, \ State, \ 2, \ d_2\rangle, \\ \ \ \langle \texttt{opened}, \ State, \ 5, \ d_3\rangle\ ] \end{array}$$

$$P_2 = \begin{array}{l} [\ \langle \texttt{doors}, \ Machine, \ 1, \ d_4\rangle, \\ \ \ \langle \texttt{closed}, \ State, \ 2, \ d_5\rangle, \\ \ \ \langle \texttt{opened}, \ State, \ 6, \ d_6\rangle, \\ \ \ \langle \texttt{locked}, \ State, \ 9, \ d_7\rangle\ ] \end{array}$$

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) (line 4). The main **while**-loop then iterates over the projections P_1 and P_2 in parallel, distributing definition labels over the A, D and I sets that will make up the matching (lines 6–17). If a name occurs unchanged in both src_1 and src_2, an additional type check prevents that entities in different categories are matched (lines 12–15).

The result of matching is a triple $M = \langle A, D, I\rangle$, where $A \subseteq L_{m_2}$ contains new elements in m_2, $D \subseteq L_{m_1}$ contains elements removed from m_1, and $I \subseteq L_{m_1} \times L_{m_2}$ represents identified entities, where L_{m_1} and L_{m_2} are labels of elements in m_1 and m_2, respectively.

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

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

Next we explain how the matching result is used for differencing nodes.

Differencing

The heavy lifting of TMDIFF is realized by the diffNodes function. It is shown in Fig. 10. It receives an existing entity as the current context (ctx), the two elements to be compared (m_1 and m_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 m_1 and m_2. diffNodes assumes that both m_1 and m_2 are of the same class (line 3). The algorithm then loops over all fields that need to be differenced (lines 5–17). Fields can be of four kinds: primitive (lines 6–7), containment (lines 8–12), reference (lines 13–14) or list (lines 15–16). 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

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

```
1   list[Operation] diffNodes(obj ctx, obj m1, obj m2, Path p,
2       Matching M) {
3     assert m1.class = m2.class;
4     Δ = []
5     for (f ←m1.class.fields) {
6       if (f.isPrimitive && m1[f] ≠ m2[f])
7         Δ += [new SetPrim(ctx, p+[f], m2[f])];
8       else if (f.isContainment)
9         if (m1[f].class = m2[f].class)
10          Δ += diffNodes(ctx, m1[f], m2[f], p+[f], M)
11        else
12          Δ += [new SetTree(ctx, p+[f], build(m2[f], M))]
13      else if (f.isReference && M⁻¹[m2[f]] ≠ m1[f])
14        Δ += [new SetRef(ctx, p+[f], M⁻¹[m2[f]])]
15      else if (f.isList)
16        Δ += diffLists(ctx, m1[f], m2[f], p+[f], M)
17    }
18    return Δ
19  }
```

Fig. 10 Differencing nodes

interesting bit happens when differencing reference fields. References are compared via the matching M, highlighted in Fig. 10.

In order to know whether two references are "equal", diffNodes performs a reverse lookup in M on the reference in m_2 (line 13). 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. 8) to contain entries for all newly created model elements. As a result, the reverse lookup (line 14) is always well-defined. Either we find an already existing element of m_1, or we find a element created as part of m_2, highlighted in Fig. 10.

3.3 Implementation in RASCAL

We have implemented TMDIFF in RASCAL, a functional programming language for metaprogramming and language workbench for developing textual DSLs [16]. The code for the algorithm, the application to the example state machine language, and the case study can be found on GitHub.[3]

Since RASCAL is a textual language workbench [7], 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. 4 is labeled:

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

Because RASCAL is a functional programming language, all data are immutable and first-class references to objects are unavailable. Therefore, 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.

In prior work [43], we have evaluated TMDIFF on the version history of *file format specifications* written in Derric, a real-life DSL that is used in digital forensics analysis [37]. We found that TMDIFF reliably computes small deltas between consecutive versions of the Derric specifications of JPEG, GIF, and PNG.

4 RMPatch: generic run-time model patching

4.1 Overview

The previous section described the TMDIFF algorithm to obtain model-based deltas from textual source files. Here we introduce RMPATCH, a generic architecture to apply these deltas to run-time models that drive the execution of the models of a language. During interpretation of such a model, users edit the textual model using a live programming environment that embeds TMDIFF for generating deltas for successive model versions, as shown in Fig. 11 on the left. These edit scripts are applied by RMPATCH to migrate the model as part of the running program to reflect the new version of the source code, as shown in Fig. 11 on the right. Together TMDIFF and RMPATCH provide a foundation for the design and implementation of live programming environments, where textual models can be edited while they are executing.

In order to provide a unified approach for recording and replaying model differences, we record a run-time history of events such as user interactions and changes in the source code as edit operations on the run-time model. This history can be used for implementing "undo," persisting application state (cf. event sourcing), and back-in-time debugging. When the developer edits a textual model and saves a modified version, the programming environment applies TMDIFF to the current and the previous version of the textual model. It

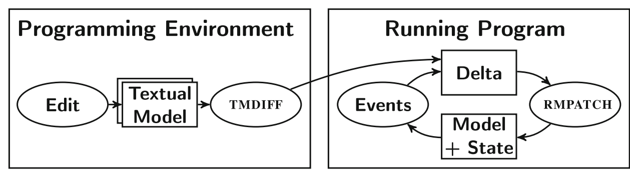


Fig. 11 Approach: using TMDIFF and RMPATCH for live programming with textual models

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

then passes the resulting delta to RMPATCH, which pauses the interpreter, applies the delta to the run-time model, possibly migrating run-time state, and continues the interpreter. Similarly, we also represent the effects of other events as deltas, e.g., resulting from a user pressing a button or a sensor firing. In Fig. 11, the oval "events" represents these cases.

4.2 Models at run time

Live programming environments enable adapting models at run time as text. Specifically, a model is an instance of a static metamodel of a language represented by an ASG, which is obtained from text through parsing and name resolution. RMPATCH requires that a model interpreter is implemented in an object-oriented language, like Java. In particular, it requires reflection for interpreting executable deltas that create objects and assign values to fields. The interpreter executes a model as a *run-time model*, an instance of a *run-time metamodel*, which extends the static metamodel of the language by adding additional attributes and relations to model run-time state, and methods that implement behavior.

For instance, a state machine can be executed by interpreting incoming events and updating a current state attribute. In between such transitions, the run-time model may need to be migrated, however, because, in a live programming environment, the source code of the state machine may have changed in the meantime. At dedicated points in the execution, the interpreter must check for pending deltas (as produced by TMDIFF), and if there are any, apply them to the run-time model, before continuing execution.

4.3 Applying deltas at run time

The deltas produced by TMDIFF are converted to run-time edit operations that can be evaluated against an instance of the run-time metamodel. Every change computed by TMDIFF can be mapped to a change at run time, because the model of the source is subsumed by the run-time model. Applying a run-time delta contributes a sequence of atomic edits to the run-time history of the running program. The edit operations produced by TMDIFF, however, are unaware of any additional state maintained in the run-time models. For avoiding information loss and invalid run-time states, RMPATCH can be extended with custom state migrations. Migration effects are represented as model edits too, making them part of the run-time history.

Recall that TMDIFF produces edit scripts as shown in Fig. 7:

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

Such a script is represented as a list of edits, such as `create`, `setTree`, `insertTree` and `insertRef`. In addition to these four, TMDIFF generates `delete`, `setPrim`, `remove`, `insertRef` and `setRef` operations. `Create` and `delete` are global operations, creating or deleting objects from the model, respectively. The other, relative operations traverse a path through the features of their *owner object*, the object operated on (e.g., d_7, d_2, or d_1), and modify the traversed field accordingly. For instance, the last operation in the edit script above inserts state d_7 in the machine's (d_1) list of states at index 2.

The edit operations `setTree` and `insertTree` take trees as arguments. Java makes no distinction between a tree argument's containment references and cross-references and encodes both as object references. We therefore flatten tree operations to a sequence of `create`, `setPrim`, `setRef` and `insertRef` operations. As a result, RMPATCH only implements these operations, and `delete` and `remove`.

Owner objects are represented using opaque identities used internally by TMDIFF. RMPATCH maintains an `objectSpace` table that maps these identities to Java objects. The `create` and `delete` operations, respectively, add and remove objects in this table. Since the identities are not stable across versions of a model, RMPATCH uses the TMDIFF matching (see Sect. 3.2) information to adjust the object space to reflect the situation after the edit operations have been applied.

Applying the edit operations to the run-time model is implemented using the Visitor pattern [10]. A base visitor defines visit methods for each type of edit operation and modifies the current model according to the semantics of the operation. When an edit has been applied, it is added to the global history object to support undo and replay.

The application of edit operations to a run-time model is unaware of invariants concerning the run-time state extensions of that model. Naively applying a TMDIFF delta to the run-time model of a DSL program might bring its execution in an inconsistent state. For instance, in the case of state machines, what happens if the current state is removed? What happens if the last remaining state is removed? These questions cannot be answered in a generic, language independent way. We therefore allow the base visitor to be extended with custom state migration logic to address such questions. If such additional migration steps are realized as edit operations as well, they can also be added to the global application history, to ensure that undo and replay maintain consistency.

The next section describes how these technique have been applied in the development of a live programming environment for the state machine language of Sect. 3.

5 Case study: live state machine language

5.1 Overview

Here we present a case study based on the simple state machine language (SML) used as the running example in Sect. 3. We have used both TMDIFF and RMPATCH to obtain a live programming environment for SML, called LiveSML. The static and run-time metamodels of SML are shown in Fig. 12.

The run-time model (Fig. 12b) can be seen as an extension of the static metamodel (Fig. 12a); it includes all the attributes and relations of the static model. However, to represent run-time state, there are additional attributes and relations that do not exist in the static metamodel. For instance, run-time machines (Mach objects) have a state field, representing the current state. Furthermore, the State objects are extended with a count field, indicating how many times this state has been visited.

LiveSML consists of two application components, shown in the top row of Fig. 13. On the left, Fig. 13a shows the programming environment of LiveSML, which consists of an Eclipse-based IDE for editing state machines, implemented in RASCAL. The editor shows the $Doors_1$ state machine.

On the right, Fig. 13b shows the execution of $Doors_1$ as an interactive GUI. The user can click buttons corresponding to events defined in the state machine. The main window shows a textual rendering of the state machine in tabular form. An asterisk indicates which state is the current one, and the column marked with the pound symbol indicates how many times a state has been visited. The bottom row shows the actual $Doors_1$ state machine models. Figure 13c shows the static state machine model that represents the textual source code of $Doors_1$ shown in the editor. Figure 13d shows the same state machine, represented as a dynamic model that is executing at run time, which is shown in the GUI.

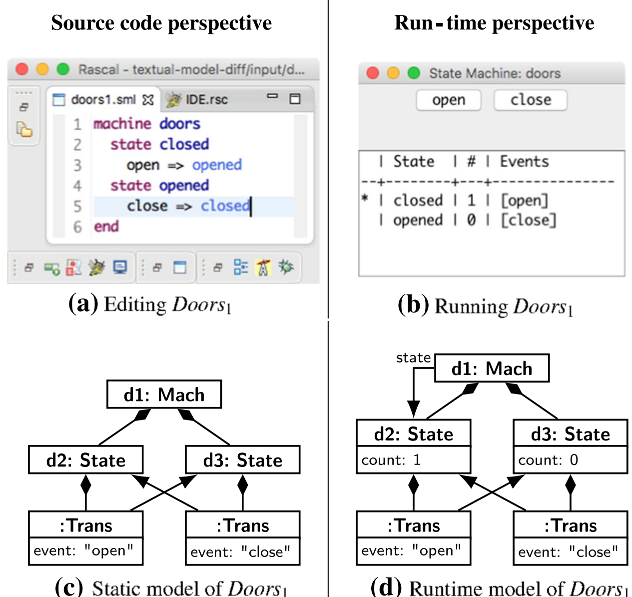


Fig. 13 LiveSML: the *left* shows the source code perspective with the IDE at the *top* and the static model at the *bottom*. The *right* shows the run-time perspective with the state machine GUI at the *top*, and the (extended) run-time model at the *bottom*

When a developer edits a textual model and saves a modified version, the programming environment applies TMDIFF to the current and the previous version of the textual model. It then passes the resulting delta to the executing program that embeds RMPATCH. Similarly, when the user triggers an event, the program calculates its own delta for updating its model elements. As a result, runtime model transformations result either from textual model edits or user-level application events.

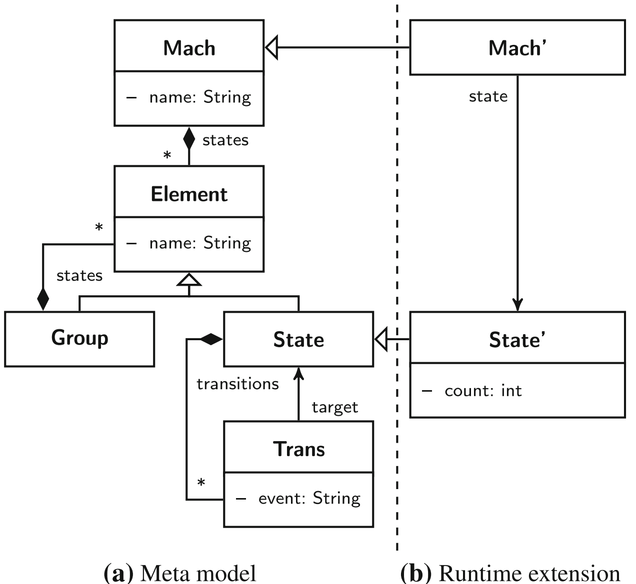


Fig. 12 Static and run-time metamodel of SML

5.2 Migrating domain-specific run-time state

Since the deltas produced by TMDIFF only take the static metamodel of the source into account, the generic RMPATCH system needs to be extended to support dealing with the *state* and *count* attributes. Note that in most cases, RMPATCH will simply leave these attributes intact, but in special cases, the outcome would lead to an inconsistent state of the execution.

We define domain-specific state migration logic by extending the ApplyDelta visitor provided by RMPATCH, as shown in Fig. 14. The class ApplyDelta defines a visit method for each kind of edit supported by RMPATCH. For LiveSML, we address the following cases:

```
1  class MigrateSML extends ApplyDelta {
2    private Mach machine; //run-time model to migrate
3
4    @Override
5    public void visit(Create create) {
6      super.visit(create);
7
8      Object x = create.getCreated(this);
9      if (x instanceof Mach) { //new machine
10       this.machine = (Mach) x;
11     }
12     else if (x instanceof State) { //new state
13       Edit e = new SetPrim(reverseLookup(x),
14                 new Path(new Field("count")), 0);
15       e.accept(this);
16     }
17   }
18
19   @Override
20   public void visit(Insert insert) {
21     super.visit(insert);
22
23     Object owner = insert.getOwner(this);
24     if (machine != null && machine.state == null
25          && owner == machine) {
26       // Added a group or state to a machine
27       // without a current state.
28       goToInitialState();
29     }
30   }
31
32   @Override
33   public void visit(Delete delete) {
34     super.visit(delete);
35
36     Object x = delete.getDeleted(this);
37     if (machine != null && x == machine.state) {
38       // Deleted the current state.
39       goToInitialState();
40     }
41   }
42
43   private void goToInitialState(){
44     State s = machine.findInitial();
45     Edit e1 = new Set(reverseLookup(machine),
46        new Path(new Field("state")), s);
47     e1.accept(this); //Set the current state.
48
49     if (s != null){
50       Edit e2 = new Set(reverseLookup(s),
51          new Path(new Field("count")), s.count+1);
52       e2.accept(this); //Increment current state count.
53     }
54   }
55 }
```

Fig. 14 MigrateSML extends ApplyDelta for SML state migration

- *Creation of a new machine* Initially there is no machine because we start with an empty object space. We store a reference to the machine when it is first created (lines 9 and 10).
- *Creation of a new state* The *count* attribute is initialized to 0 (lines 12–15).
- *Insertion of an element in an uninitialized machine* When a state or group is inserted into a machine that has no current state (lines 24–29), it is initialized to the *initial state* (lines 43–54). The initial state is the first state in the textual model.
- *Deletion of the current state* When a machine's current state is deleted (lines 36–37), it is reinitialized to the initial state (lines 43–54).

Each domain-specific migration is represented using edit operations. For each required side effect, new edit objects are created. For instance, initializing the *count* field of a new state to 0 is enacted by a SetPrim edit, anchored at the new state, with a path to field "count". Applying these operations through the extended visitor (MigrateSML) adds them to the application history of LiveSML.

5.3 Evolving and using state machines with LiveSML

The key point of LiveSML is that state machines can be edited and used at the same time. In a sense, the source and run-time models coevolve in lockstep: changes in the code are interleaved with user events—both transform the run-time model using deltas. To illustrate this coevolution, we present a prototype live editing scenario with LiveSML.

Figure 15 shows its general time line. The top row shows five successive versions of the state machine definition, starting in the version where there is no state machine at all (∅). The bottom row shows successive states of the executing state machine. Some state changes are triggered by source changes (e.g., from s_0 to s_1), while others result from user interactions (e.g., s_2–s_3).

The details of the application state transitions are listed in Table 1. The first two columns indicate the start source model and run-time model state. The third column ("event") captures what happened ("saving" or "clicking an event button"). Each event causes a sequence of edits δ_i to be applied to the run-time model. Edits correspond directly to the operations generated by TMDIFF. One additional operation (rekey) is used to realign the internal object identities of the run-time model with the opaque identities used by TMDIFF; this operation is needed because the TMDIFF identities are not stable across revisions. The last column shows the origin of the edit operations: an edit can originate from a TMDIFF delta, a migration side effect (as described in Sect. 5.2), or a user action. The sequence of δ_i ($i \in 1 \ldots 41$) represents the full history of run-time model transformations.

Finally, Table 2 shows, yet again, the sequence of source models and program states of the LiveSML session—this time showing both the editor and the runtime GUI. From left to right, the upper row shows states s_0–s_3, and the bottom row s_4–s_7. An empty cell indicates that nothing has changed in the editor with respect to the previous state.

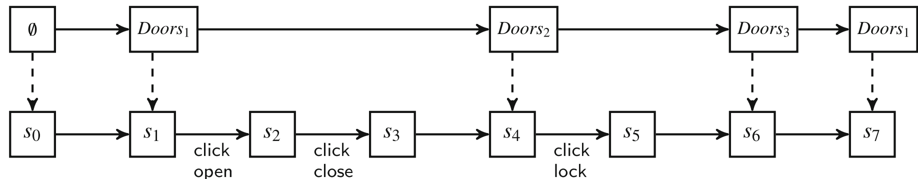


Fig. 15 Interleaved coevolution of models $Doors_n$ and application run-time states S_n over time

Table 1 Interleaved coevolution of models $Doors_n$ and run-time states S_n over time

Model	State	Event	Edit operation		Origin
∅	s_0	Save $Doors_1$	δ_1	**create** lang.sml.runtime.State d2	TMDIFF ∅ $Doors_1$
			δ_2	d2.count = 0	Side effect
			δ_3	**create** lang.sml.runtime.State d3	
			δ_4	d3.count = 0	Side effect
			δ_5	**create** lang.sml.runtime.Mach d1	
			δ_6	d2 = State(name("closed"),[Trans("open",d3)])	
			δ_7	d3 = State(name("opened"),[Trans("close",d2)])	
			δ_8	d1 = Mach(name("doors"),[d2,d3])	
			δ_9	d1.state = d2	Side effect
			δ_{10}	d2.count = 1	Side effect
$Doors_1$	s_1	Click *open*	δ_{11}	d1.state = d3	User action
			δ_{12}	d3.count = 1	
$Doors_1$	s_2	Click *close*	δ_{13}	d1.state = d2	User action
			δ_{14}	d2.count = 2	
$Doors_1$	s_3	Save $Doors_2$	δ_{15}	**create** lang.sml.runtime.State d7	TMDIFF $Doors_1$ $Doors_2$
			δ_{16}	d7.count = 0	Side effect
			δ_{17}	d7 = State(name("locked"),[Trans("unlock",d2)])	
			δ_{18}	**insert** d2.transitions[1] = Trans("lock",d7)	
			δ_{19}	**insert** d1.states[2] = d7	
			δ_{20}	**rekey** d1 → d4	
			δ_{21}	**rekey** d2 → d5	
			δ_{22}	**rekey** d3 → d6	
$Doors_2$	s_4	Click *lock*	δ_{23}	d4.state = d7	User action
			δ_{24}	d7.count = 1	
$Doors_2$	s_5	Save $Doors_3$	δ_{25}	**create** lang.sml.runtime.Group d11	TMDIFF $Doors_2$ $Doors_3$
			δ_{26}	d11 = Group("locking",[d6])	
			δ_{27}	**remove** d4.states[2]	
			δ_{28}	**insert** d4.states[2] = d0	
			δ_{29}	**rekey** d4 → d8	
			δ_{30}	**rekey** d5 → d9	
			δ_{31}	**rekey** d6 → d10	
			δ_{32}	**rekey** d7 → d12	
$Doors_3$	s_6	Save $Doors_1$	δ_{33}	**remove** d8.states[2]	TMDIFF $Doors_3$ $Doors_1$
			δ_{34}	**remove** d9.transitions[1]	
			δ_{35}	**delete** d11	
			δ_{36}	**delete** d12	
			δ_{37}	d13.state = d9	Side effect
			δ_{38}	d9.count = 3	Side effect
			δ_{39}	**rekey** d8 → d13	
			δ_{40}	**rekey** d9 → d14	
			δ_{41}	**rekey** d10 → d15	

Table 2 Sequence of screen shots of LiveSML's programming environment (top) and running application (bottom) while in application state s_i ($i \in 0, \dots, 7$) of the interactive session with LiveSML

<table>
<tr><th>s_0</th><th>s_1</th><th>s_2</th><th>s_3</th></tr>
<tr><td>Rascal - textual-model-diff/input/...
doors1.sml IDE.rsc
1</td><td>Rascal - textual-model-diff/input/...
doors1.sml IDE.rsc
1 machine doors
2 state closed
3 open => opened
4 state opened
5 close => closed
6 end</td><td></td><td></td></tr>
<tr><td>State machine</td><td>State Machine: doors
open close
| State | # | Events
--+--------+---+--------------
* | closed | 1 | [open]
| opened | 0 | [close]</td><td>State Machine: doors
open close
| State | # | Events
--+--------+---+--------------
| closed | 1 | [open]
* | opened | 1 | [close]</td><td>State Machine: doors
open close
| State | # | Events
--+--------+---+--------------
* | closed | 2 | [open]
| opened | 1 | [close]</td></tr>
<tr><th>s_4</th><th>s_5</th><th>s_6</th><th>s_7</th></tr>
<tr><td>Rascal - textual-model-diff/input/...
doors1.sml IDE.rsc
1 machine doors
2 state closed
3 open => opened
4 lock => locked
5 state opened
6 close => closed
7 state locked
8 unlock => closed
9 end</td><td></td><td>Rascal - textual-model-diff/input/...
doors1.sml IDE.rsc
1 machine doors
2 state closed
3 open => opened
4 lock => locking.locked
5 state opened
6 close => closed
7 locking
8 {
9 state locked
10 unlock => closed
11 }
12 end</td><td>Rascal - textual-model-diff/input/...
doors1.sml IDE.rsc
1 machine doors
2 state closed
3 open => opened
4 state opened
5 close => closed
6 end</td></tr>
<tr><td>State Machine: doors
open lock close
unlock
| State | # | Events
--+--------+---+--------------
* | closed | 2 | [open, lock]
| opened | 1 | [close]
| locked | 0 | [unlock]</td><td>State Machine: doors
open lock close
unlock
| State | # | Events
--+--------+---+--------------
| closed | 2 | [open, lock]
| opened | 1 | [close]
* | locked | 1 | [unlock]</td><td></td><td>State Machine: doors
open close
| State | # | Events
--+--------+---+--------------
* | closed | 3 | [open]
| opened | 1 | [close]</td></tr>
</table>

We now briefly describe how each run-time model state s_n in the sequence results from textual model edits and user actions.

- s_0 The application starts and the initial model is $\emptyset$. Both the editor and GUI are empty.
- s_1 *Doors*$_1$ is entered into the editor and saved. In response, the environment computes the difference TMDIFF $\emptyset$ *Doors*$_1$. As a result, the GUI shows the execution of *Doors*$_1$. Both state count attributes are initialized to zero (δ_2 and δ_4). The machine's initial state is *closed* (marked by *), and its count is set to one (δ_9 and δ_{10}).
- s_2 The user clicks button *open*, which triggers the transition and produces δ_{11} and δ_{12}.
- s_3 The user clicks button *close*, which triggers the transition and produces δ_{13} and δ_{14}.
- s_4 The model is modified such that it becomes *Doors*$_2$. In response, the environment computes the difference between *Doors*$_1$ and *Doors*$_2$. The *count* attribute of the *locked* state is initialized to zero (δ_{16}). The UI now also displays buttons for the *lock* and *unlock* events.
- s_5 The user clicks button *lock*, which triggers the transition and produces operations δ_{23} and δ_{24}.
- s_6 The model is modified such that it becomes *Doors*$_3$. In response, the environment computes the difference between *Doors*$_2$ and *Doors*$_3$. This time, there are no migration side effects because the change has no semantic effect: grouping is just a scoping mechanism.

– s_7 Finally, the model is modified such that it becomes *Doors*$_1$ again. As a result of applying the differences, the current state *locked* is removed and therefore the current state is reinitialized to the first state *closed* (δ_{37}). Accordingly, its *count* is set to three (δ_{38}). Note that the buttons *lock* and *unlock* have been removed from the UI since no such events exist anymore.

The sequence of states of this LiveSML session shows the fine-grained interleaving of edit operations originating from different sources. The execution of the state machine adapts to both user events and changes in the source code. As such, LiveSML provides a very fluid developer experience. Long edit-compile cycles are completely eliminated.

6 Discussion and related work

This paper presents an approach for live programming environments for textual DSLs that builds on two reusable components: TMDIFF and RMPATCH. We reflect on limitations, challenges and future work, and discuss related work.

6.1 Toward live domain-specific languages

Live DSLs aim for a low representation gap between domain, notation and run time. Users can adapt run-time models directly from the textual source. We assume that the run-time metamodel extends the static language metamodel, such as is the case in LiveSML. This design choice facilitates applying changes in the source code to the running program. The assumption does not hold in general, however. For instance, imperative languages have more complex mappings between code and execution. Such languages therefore offer less direct affordances over a program's execution, breaking the continuous link between the mental model of the programmer, the code and the running program.

Edit scripts are commonly used to encode model differences between versions of models representing the abstract syntax of a language. Edit scripts precisely encode *what* changed and in which order, but not *why* these effects happen. Typically, language semantics refers to a formal definition that does include the precise causal relationships from which these run-time changes result, which also enables formal proofs. In our approach, the behavioral evolution of executing models is influenced by the way model differences are computed. When entities are not detected as "the same" between versions, the corresponding run-time objects will be removed or added, even if this was not the behavior intended by the user of the modeling language. This problem is not unique to our application of TMDIFF, since any differencing algorithm will have to use heuristics to match model elements. We hypothesize, however, that in the context of live programming where immediacy of feedback is paramount, changes tend to be small and local, reducing the risk of unintuitive matchings.

One question is whether replacing TMDIFF by an alternative algorithm would provide a better programmer experience. For instance, SiDiff [15,36], DSMDiff [24] or EMF-Compare [6] may result in a more accurate matchings for specific circumstances. SiDiff in particular would be a candidate since it is independent from any kind of scoping rules used to create references between model elements. SiDiff can be configured to make the algorithm perform better based on certain language features. Unfortunately, adjusting the weights used in comparing language features often requires substantial empirical testing [17].

The question is whether similarity-based heuristics would offer more predictable differences, and as a result more predictable run-time adaptation. Our hypothesis is that TMDIFF has the benefit that its mechanism for identifying model elements stays close to the textual source representation of a model, which is precisely the material the modeler is manipulating. Comparing alternative differencing approaches in terms of predictability and run-time performance is part of future work.

Our experience in using TMDIFF and RMPATCH shows that migrating run-time state is complex. Even for a relatively simple language like LiveSML, the extensions of RMPATCH to migrate state must account for many possible transformation scenarios. Since edit operations are applied in sequence, one must make careful assumptions about the existence or absence of objects and references. The key question is then whether the correct interleaving of migration edits with the original edits produced by TMDIFF could be automatically derived. In future work, we plan to address this challenge by separately modeling and maintaining migration scenarios that abstract from underlying edits, and use dependency analysis to derived possible orderings of run-time model modifications.

Assessing whether RMPATCH scales to larger systems requires additional case studies on real-world live DSLs, in particular those whose source and run-time metamodels differ more substantially than in the case of LiveSML. To investigate this question further, we plan to apply RMPATCH to Micro-Machinations, a visual language and execution engine that enables game designers to adapt a game's mechanics while it is running [42]. Its live programming environment is called Mechanics Design Assistant (MeDeA) [41].

The run-time metamodel of Micro-Machinations adds a new level of dynamic instantiation: at runtime there are "instance" level models which are not directly represented by textual source code, but which depend on source-defined entity definitions. Such languages require a pipeline of coupled transformations between source and run-time. The question is how modification effects propagate in a well-

defined way. This problem is not unlike migrating objects after a change in class (e.g., in Smalltalk), or database migration upon schema change. In fact, these kinds of migrations are instances of the general class of *coupled transformations* [19] where a transformation of one model induces a "coupled" transformation on another (possibly over a different metamodel). Further research is needed to formalize run-time patching presented here using this framework. This could help to precisely delineate the scope and limitations of RMPATCH-like run-time adaptation.

Reversible transformations support features for programming environments such as undoing edits, rollback, restoring system states, replaying and debugging. RMPATCH operations can be augmented with extra information to make every edit operation—and thus complete edit scripts—reversible. The question is to what extent such features can be supported by generic, reusable components. Although it is clear how to "unapply" edit operations on the run-time model, performing this same operation on the textual source code requires more advanced machinery, such as origin tracking, source code formatting and reversing source-to-source transformations.

At this time, TMDIFF and RMPATCH offer no special support for model merging, which, for instance, would be interesting for hypothetical exploration of dynamic what-if scenarios. Further research is needed to investigate how different deltas produced by TMDIFF can be combined for this purpose and how to resolve merge conflicts at runtime.

6.2 Limitations of TMDiff

Unlike RMPATCH, the TMDIFF algorithm can be used independently. In this section, we identify a number of limitations of TMDIFF as a separate component and discuss 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 matchings, 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 without any further change, TMDIFF will not detect it as such. 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., [35]. A related problem is that TMDIFF will always see renames as an addition and removal of an entity. In general, edit scripts consisting of long sequences of atomic operations are hard to understand. However, user-level composite operations such as renaming and more complex refactorings can be detected in existing sequences of atomic operations, e.g., using the approach proposed by Langer et al. [21], or the rule-based semantic lifting approach proposed by Kehrer et al. [14].

6.3 Related work

The key contribution of this paper intersects two areas of related work: model differencing and dynamic adaptation of models at runtime. Below we discuss important related work in both these areas.

6.3.1 Model differencing

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 [33]. 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 [17,33]. 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 [33]. 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 based on Alanen and Porres' seminal work [1]. The identification map 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., [45]).

TMDIFF's differencing strategy resembles the model merging technique used Ensō [39]. 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 container (e.g., a set or list). 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. 4c, where the `locked` state moved from the global scope 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 [18]. It integrates HUTN [32], 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.

Lin et al. [24] describe DSMDiff, a signature-based differencing approach which is intended specifically for Domain-Specific Modeling Languages. DSMDiff uses a signature-based matching over node and edge model elements, augmented by structural matching when the signature-based matching produces multiple matching candidates.

Maoz et al. [26] propose *semantic differencing*, an approach that defines *diff operators* for comparing two models where the resulting differences are presented as a set of semantic *diff witnesses*, instances of the first model that are not instances of the second. These instances are concrete examples explaining how the models differ. Maoz and Ringert [25] relate syntactic changes to semantic witnesses by defining necessary and sufficient sets of change operations.

Langer et al. present a general approach for semantic differencing that can be customized for specific modeling languages. This approach is based on the behavioral semantics of a modeling language [20]. Two versions of a model are executed to capture execution traces that represent its semantic interpretation. Comparing these traces then provides a "semantic" interpretation of the difference between the two versions. In contrast, our approach starts at the opposite end: instead of using execution traces to explain syntactic differences, we use syntactic differences to drive the execution in the first place.

Cicchetti et al. [4] propose a representation of model differences which is model based, transformative, compositional and metamodel independent. Differences are represented as models that can be applied as patches to arbitrary models. Although no special extension points are offered for supporting run-time state migrations, the model-based differences themselves could be used to represent them.

6.3.2 Dynamic adaptation

"Models at run time" is a well-researched topic, as, for instance, witnessed by the long running workshop on Models@run.time [12]. Executable modeling can be considered a subdomain of models at run time, where a software system's execution is defined by a model interpreter. Executable modeling was pioneered in the context of the Kermeta system [5,30]. Kermeta is also the basis for recent work on omniscient debugging features for xDSMLs [2]. Omniscient debuggers allow the execution of a program or model to be reversed and replayed. This work can be positioned on an orthogonal axis of "liveness," where the focus is on providing better feedback through time travel. We consider our delta-based approach to be a fruitful ground for further exploration of such features. In the LiveSML case study, we already have implemented a reversible history of application state. However, a particular challenge will be to apply reversed edits back to the source code of a DSL program.

Models at run time in general are often motivated from the angle of dynamic adaptation. For instance, Morin et al. [29] describe an architecture to support adaptation at run time through aspect weaving. However, this work focuses on adapting behavior and dynamically selecting alternative variants of behavior, rather than changing the run-time models themselves.

The specific requirements for run-time metamodeling are explored by Lehmann et al. [22]. The authors present a process to identify the core run-time concepts occurring in run-time models. In particular, they propose to identify possible model adaptations at run time, to explicitly address potential run-time consistency issues. In our case, we allow any kind of modification, but leave the door open to implement arbitrary run-time state migration policies.

RMPATCH requires the run-time metamodel to be an "extension" of the static metamodel. This relation is similar to the concept of "subsumption" in description logics [27]. Although we have not yet explored this link in more detail, it would allow formal checking of whether a run-time metamodel is suitable for live patching. Another assumption underlying RMPATCH is that it should be possible to pause the model interpreter at a stable point in the execution in order to apply the runtime modifications. This is related to the concept of quiescence explored in the area of dynamic software updating [44].

7 Conclusion

Live programming promises to improve developer experience through immediate and continuous feedback. These

benefits have not yet been explored from the perspective of executable domain-specific modeling languages. In this paper, we have described a framework for developing "live textual languages," based on a metamodeling foundation. Our framework consists of two components.

First, we presented TMDIFF, a novel model differencing algorithm, based on textual differencing and origin tracking. 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. TMDIFF is a fully language parametric approach to textual model differencing. A prototype of TMDIFF has been implemented in the RASCAL metaprogramming language [16].

The second component, RMPATCH, represents an architecture for dynamically adapting run-time models which encode the execution of the model. RMPATCH receives model deltas from TMDIFF and evolves the execution accordingly. To avoid information loss and invalid run-time states, RMPATCH can be extended to define custom, language-specific migration policies. RMPATCH is used in the development of a live state machine DSL, which allows simultaneous editing *and* using of state machine definitions.

To the best of our knowledge, this paper is the first work connecting the worlds of model differencing and dynamic adaptation of models at run time. Nevertheless, some important directions for further research remain. The most important directions are formalizing the relation between static metamodel and (extended) run-time metamodel of a DSL, investigating how dependencies between edit operations can be inferred and used to (re)order their application, and determining how to separately model and maintain run-time state migration scenarios at a higher level of abstraction. Ultimately, we expect that delta-based run-time adaptation provides a fertile foundation for developing live programming support for executable DSLs.

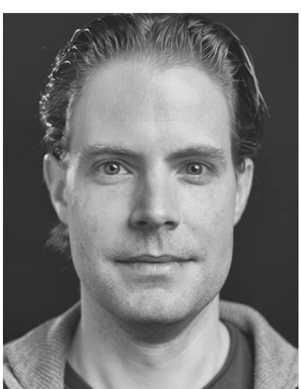
Acknowledgements We thank the reviewers for their insightful comments that helped improve this paper.

References

1. Alanen, M., Porres, I.: Difference and union of models. In: Stevens, P., Whittle, J., Booch, G. (eds.) «UML» 2003—The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference, San Francisco, CA, USA, October 20–24, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2863, pp. 2–17. Springer, (2003)
2. Bousse, E., Corley, J., Combemale, B., Gray, J., Baudry, B.: Supporting efficient and advanced omniscient debugging for xDSMLs. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, pp. 137–148. ACM, (2015)
3. Brun, C., Pierantonio, A.: Model differences in the eclipse modeling framework. UPGRADE Eur. J. Inform. Prof. **9**(2), 29–34 (2008)
4. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: Model patches in model-driven engineering. In: Ghosh, S. (ed.) Models in Software Engineering: Workshops and Symposia at MODELS 2009, Denver, CO, USA, October 4–9, 2009, Reports and Revised Selected Papers, pp. 190–204. Springer, Berlin, (2009)
5. Combemale, B., Crégut, X., Pantel, M.: A design pattern to build executable DSMLs and associated V&V tools. In: Software Engineering Conference (APSEC), 2012 19th Asia-Pacific, vol. 1, pp. 282–287. IEEE, (2012)
6. Eclipse Foundation. EMF Compare Project. https://www.eclipse.org/emf/compare/
7. Erdweg, S., van der Storm, T., Völter, M., et al.: The state of the art in language workbenches. In: Erwig, M., Paige, R.F., Van Wyk, E., (eds.) Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26–28, 2013, Proceedings. Lecture Notes in Computer Science, vol. 8225, pp. 197–217. Springer, (2013)
8. Erdweg, S., van der Storm, T., Völter, M., Tratt, L., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G., Molina, P.J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V., Visser, E., van der Vlist, K., Wachsmuth, G., van der Woning J.: Evaluating and comparing language workbenches: existing results and benchmarks for the future. Comput. Lang. Syst. Struct. **44**, Part A, 24–47 (2015). In: Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014)
9. 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, pp. 307–309, New York, NY, USA. ACM, (2010)
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley, Boston (1995)
11. 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). LNCS, vol. 5095, pp. 169–184. (2008)
12. Götz, S., Bencomo, N., France, R.: Devising the future of the Models@Run.Time workshop. SIGSOFT Softw. Eng. Notes **40**(1), 26–29 (2015)
13. 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. LNCS, vol. 8568, pp. 154–169. Springer, Berlin (2014)
14. Kehrer, T., Kelter, U., Taentzer, G.: A rule-based approach to the semantic lifting of model differences in the context of model versioning. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), pp. 163–172. (2011)
15. Kehrer, T., Kelter, U., Pietsch, P., Schmidt, M.: Adaptability of model comparison tools. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, pp. 306–309, New York, NY, USA. ACM, (2012)
16. Klint, P., van der Storm, T., Vinju, J.: Rascal: a domain-specific language for source code analysis and manipulation. In: Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM '09, pp. 168–177. (2009)

17. 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), pp. 1–6. IEEE, (2009)
18. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The epsilon transformation language. In: Theory and Practice of Model Transformations, pp. 46–60. Springer, (2008)
19. Lämmel, R.: Coupled software transformations. In: First International Workshop on Software Evolution Transformations, pp. 31–35. (2004)
20. Langer, P., Mayerhofer, T., Kappel, G.: Semantic Model Differencing Utilizing Behavioral Semantics Specifications. Springer International Publishing, Berlin (2014)
21. Langer, P., Wimmer, M., Brosch, P., Herrmannsdörfer, M., Seidl, M., Wieland, K., Kappel, G.: A posteriori operation detection in evolving software models. J. Syst. Softw. **86**(2), 551–566 (2013)
22. Lehmann, G., Blumendorf, M., Trollmann, F., Albayrak, S.: Meta-modeling runtime models. In: Models in Software Engineering, pp. 209–223. Springer, (2010)
23. Lieberman, H., Fry, C.: Bridging the gulf between code and behavior in programming. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'95), pp. 480–486. ACM Press/Addison-Wesley Publishing Co., (1995)
24. Lin, Y., Gray, J., Jouault, F.: DSMDiff: a differentiation tool for domain-specific models. Eur. J. Inf. Syst. **16**(4), 349–361 (2007)
25. Maoz, S., Ringert, J.O.: A framework for relating syntactic and semantic model differences. In: 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 24–33. (2015)
26. Maoz, S., Ringert, J.O., Rumpe, B.: A manifesto for semantic model differencing. In: Dingel, J., Solberg, A., (eds.) Models in Software Engineering: Workshops and Symposia at MODELS 2010, Oslo, Norway, October 2–8, 2010, Reports and Revised Selected Papers, pp. 194–203. Springer, Berlin, (2010)
27. McGuinness, D.L., Borgida, A.: Explaining subsumption in description logics. IJCAI **1**, 816–821 (1995)
28. Miller, W., Myers, E.W.: A file comparison program. Softw. Pract. Exp. **15**(11), 1025–1040 (1985)
29. Morin, B., Barais, O., Jezequel, J.-M., Fleurey, F., Solberg, A.: Models at runtime to support dynamic adaptation. Computer **42**(10), 44–51 (2009)
30. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In: Model driven engineering languages and systems, pp. 264–278. Springer, (2005)
31. Myers, E.W.: An $O(ND)$ difference algorithm and its variations. Algorithmica **1**(1–4), 251–266 (1986)
32. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.C.: Onstructing 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. LNCS, vol. 5301, pp. 249–263. Springer, Berlin (2008)
33. Stephan, M., Cordy, J.R.: A survey of model comparison approaches and applications. In: Hammoudi, S., Pires, L.F., Filipe, J., das Neves, R., (eds.) Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2013), pp. 265–277. SciTePress, Setúbal, (2013)
34. Tanimoto, S.L.: A perspective on the evolution of live programming. In: 1st International Workshop on Live Programming (LIVE'13), pp. 31–34. IEEE, (2013)
35. Tichy, W.F.: The string-to-string correction problem with block moves. ACM Trans. Comput. Syst. **2**(4), 309–321 (1984)
36. Treude, C., Berlik, S., Wenzel, S., Kelter, U.: Difference computation of large models. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC-FSE '07, pp. 295–304, New York, NY, USA. ACM, (2007)
37. van den Bos, J., van der Storm, T.: Bringing domain-specific languages to digital forensics. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011), pp. 671–680. ACM, (2011). Software Engineering in Practice
38. van der Storm, T.: Semantic deltas for live DSL environments. In: Proceedings of the 1st International Workshop on Live Programming, LIVE '13, pp. 35–38, Piscataway, NJ, USA. IEEE Press, (2013)
39. van der Storm, T., Cook, W.R., Loh, A.: The design and implementation of object grammars. Sci. Comput. Program. **96**, Part 4(0), 460–487 (2014). In: Selected Papers from the Fifth International Conference on Software Language Engineering (SLE 2012)
40. van Deursen, A., Klint, P., Tip, F.: Origin tracking. Symb. Comput. **15**, 523–545 (1993)
41. van Rozen, R.: A pattern-based game mechanics design assistant. In: Proceedings of the 10th International Conference on the of Foundations of Digital Games (FDG 2015). Society for the Advancement of the Science of Digital Games, (2015)
42. van Rozen, R., Dormans, J.: Adapting game mechanics with micro-machinations. In: Proceedings of the 9th International Conference on the Foundations of Digital Games (FDG 2014). Society for the Advancement of the Science of Digital Games, (2014)
43. van Rozen, R., van der Storm, T.: Origin tracking + text differencing = textual model differencing. In: Theory and Practice of Model Transformations, pp. 18–33. Springer, (2015)
44. Vandewoude, Y., Ebraert, P., Berbers, Y., D'Hondt, T.: Tranquility: a low disruptive alternative to quiescence for ensuring safe dynamic updates. IEEE Trans. Softw. Eng. **33**(12), 856–868 (2007)
45. Yang, W.: Identifying syntactic differences between two programs. Softw. Pract. Exp. **21**(7), 739–755 (1991)

Riemer van Rozen is a lecturer and researcher at the Amsterdam University of Applied Sciences (AUAS), and a Ph.D. candidate at the Software Analysis and Transformation group at Centrum Wiskunde & Informatica (CWI). Since 2011, he has collaborated with industry in several applied research projects on languages and tools that speed up development and improve software quality. His research focuses on generic solutions for domain-specific languages and live programming environments in general, and automated game design in particular. For more information, visit http://vrozen.github.io.



Tijs van der Storm is senior researcher in the Software Analysis and Transformation group at Centrum Wiskunde & Informatica (CWI), and professor in software engineering at the University of Groningen. His research focuses on improving programmer experience through new and better software languages and developing the tools and techniques to engineer them in a modular and interactive fashion. For more information, see http://www.cwi.nl/~storm.