Live Little Languages

Tijs van der Storm
storm@cwi.nl / @tvdstorm

Colloquium at university of groningen 10-6-2015
About me

- Senior researcher CWI
- Software Analysis and Transformation (SWAT)
- Currently supervising 2 PhD students
- Teach at Master Software Engineering at UvA
- Supervise around 10 MSc students per year
Rascal

- Meta programming language
- Language workbench
- w/ Paul Klint and Jurgen Vinju
- … and many others!
- http://www.rascal-mpl.org
Ensō

• Model-driven programming framework
• Composition of executable specification languages
• “App = Models + Interpreters”
• with William Cook, UT Austin
• http://www.enso-lang.org
Some recent topics

• Object Algebras
  • OOPSLA’15 (hopefully), GPCE’14, ECOOP’13

• Language workbenches
  • ICMT’15, ECOOP’14, ICMT’14, SLE’13

• Domain-specific language for digital forensics
  • ECMFA’13, ICMT’12, ICSE’11 SEIP
Reaching out

• Co-organized SDA’13, SDA’14

• Talks/workshops at
  • Code generation
  • Joy of Coding
  • Bits&Chips

• Sioux, Belastingdienst, NSpyre, NFI, Optiver…
Live little languages
Live little languages

• Live: “editing a program while it runs”
  • Continuous feedback
  • Textbook example: spreadsheet

• Little languages
  • Domain-specific languages (DSLs)
  • Notations close to problem domain
Traditional programming: aim, shoot, miss/hit, repeat

Live programming: continuous aiming, with continuous feedback
General purpose languages

Domain-specific languages
Why live little languages?

Richard Paige
@richpaige

300+ million Excel users, 9M Java users. Ouch. #sems15

18/5/15, 15:13

11 RETWEETS 6 FAVORITES

http://www.eusprig.org/horror-stories.htm
Derric: a DSL for digital forensics
Micro Machinations

Van Rozen, Dormans, Adapting game mechanics with Micro-Machinations, FDG’14
Celldown: demo

```plaintext
table grades = # A / B / C / D / E
1: | Lab | Exam | Avg | = (B2 + B3) / 2 | = round(D2)
2: | 7   | 7   | 5.  | 5.             |
3: | 3   | 7   | 5.  | 5.             |
4: | 9   | 10  | 9.5 | 10.            |

view grades = # A / B / C / D / E
1: | Lab | Exam | Avg | Grade
2: | 7   | 7   | 5.  | 5.             |
3: | 3   | 7   | 5.  | 5.             |
4: | 9   | 10  | 9.5 | 10.            |
```


repl for grades
=> 7.0
=> 7.0
>
Live QL

van der Storm, *Semantic Deltas for Live DSL Environments*, LIVE’13
Live little languages

• Trinity: runtime data and source program are interlinked

• Machinations: game adapts as game mechanics is changed

• Celldown: data, computation, test, repl etc. all in a single, integrated interface (in this case: text)

• LiveQL: source changes have immediate effect on the questionnaire
Language workbenches

• IDE + meta-language(s) to build languages + IDEs
• Power tools for building DSLs
• Our workbench of choice: Rascal
• Productivity game changer
State machine DSL in Rascal

Concrete syntax
Abstract syntax
Unparse
Desugaring
Checking
Outline
Hyperlinking
Compilation
Visual simulation
Rename refactoring
Parallel merge

524 SLOC
No liveness :-(

- Compiler typically is a **batch** transformation system
- No notion of **interacting** with the system as **whole**
- Edit/compile/restart is **slow** and loses **runtime state**
- Disconnect between **generated code** and input
Research questions

• What are generic concepts and techniques for linking and integrating program and runtime?

  • => origin tracking, bidirectional transformation

• What are generic concepts and techniques for continuous feedback?

  • => incremental updates, coupled transformations,…

• How to support building live languages in language workbenches?
Semantic Deltas

- Represent programs as models
- Execution = interpreting model + state
- Editing program => semantic delta
- Interpret the delta at runtime
- Migrate runtime state where needed
State machines

```plaintext
machine doors d1

state closed d2
    open => opened u1

state opened d3
    close => closed u2

door
```
Static meta model

- **Machine name**: str
- **State name**: str
- **Transition event**: str
- **States**: *
- **Transitions**: *
- **Out**:
Fig. 1: Three versions of a simple state machine model. Definitions and uses of states are labeled with $d_i$ and $u_j$ respectively.

2. Problem Analysis

Here we introduce the challenges of textual model differencing using a simple motivating example that is used as a running example throughout the paper. Figure 1 shows three versions of textual models of a simple state machine language. 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.

In each of the state machine models, the constructs that define entities are annotated with unique labels $d_i$. For instance, in Doors 1, the machine itself is labeled $d_1$, and both states closed and opened are labeled $d_2$ and $d_3$ respectively. Similarly, uses of states in transitions are labeled with labels $u_n$. For instance, the target state opened of the transition in closed is labeled $u_1$. In Section 3 we will see that these labels are instrumental in semantic differencing of textual models.

The goal of tmdiff is to provide meaningful differences when comparing models such as the three state machines in Fig. 1. To illustrate the problem, consider the following differences:

- In Doors 2, the state opened has a new label $u_2$.
- In Doors 3, the state locked is part of the locking group.
- In Doors 3, the transition close has a new label $u_2$.

These differences are encoded in the static model as follows:

```
(machine doors d1
 state closed d2
   open => opened u1
 state opened d3
   close => closed u2
 end)
```
Runtime meta model

- Machine name: str
- State name: str
- visited: int
- Trans event: str
- currentState
- states
- transitions
- out
Fig. 1: Three versions of a simple state machine model. Definitions and uses of states are labeled with $d_i$ and $u_j$ respectively.

**Problem Analysis**

Here we introduce the challenges of textual model differencing using a simple motivating example that is used as a running example throughout the paper. Figure 1 shows three versions of textual models of a simple state machine language. 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.

In each of the state machine models, the constructs that define entities are annotated with unique labels $d_n$. For instance, in Doors 1, the machine itself is labeled $d_1$, and both states closed and opened are labeled $d_2$ and $d_3$ respectively. Similarly, uses of states in transitions are labeled with labels $u_n$. For instance, the target state opened of the transition in closed is labeled $u_1$. In Section 3 we will see that these labels are instrumental in semantic differencing of textual models.

The goal of tmdiff is to provide meaningful differences when comparing models such as the three state machines in Fig. 1. To illustrate the problem, consider the following state machine:

```
machine doors d1
  state closed d2
    open => opened u1
  state opened d3
    close => closed u2
end
```

### Runtime model

```
:Trans event: “open”
:Trans event: “close”
```
The grouping construct acts as a scope: it allows different states to coexist in the same state machine model.

Doors: locked

machine doors d1
state closed d2
  open => opened u1
state opened d3
  close => closed u2
end

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
The grouping construct acts as a scope: it allows different grouping features of the language: the locked state in

machine doors d1
  state closed d2
    open => opened u1
  state opened d3
    close => closed u2
end

and

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

create State d7
  d7 = State("locked",[Trans("unlock",d2)])
  d2.out[1] = Trans("lock",d7)
  d1.states[2] = d7
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]
(b) tmdiff
Doors 2
Doors 3

Fig. 3: tmdiff differences between Doors $i$ and Doors $i+1$ ($i \in \{2, \ldots, n\}$). In Section 3 we will introduce a novel matching strategy that constructs such mappings.

TMDiff deltas are imperative edit scripts that consist of edit operations. Edit operations include creating and removing of AST nodes, assigning fields, and inserting or removing elements from collections. Figure 3 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. 3a captures the difference between source version Doors 1 and target version Doors 2. It begins with the creation of a new state $d_7$. States have primitive fields, id and a collection of transitions called out. For state $d_7$ these are respectively initialized with its name locked and with a new transition for event "unlock". Pre-existing state $d_2$ in Doors 1 which is mapped to $d_5$ in Doors 2 receives a new transition for event "lock" that transitions to the new state $d_7$. Finally, then new state $d_7$ is inserted at index 2 of the collection of states of the machine $d_1$ in Doors 1 which is mapped to $d_4$ in Doors 2.

The edit script introducing the grouping construct locking between Doors 2 and Doors 3 is shown in Figure 3b. It starts with the creation of a new group $d_{11}$. Its initialization sets id to "locking" and states to a list with a single 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 it is replaced by the group $d_{11}$.

Although the syntactic structure of the target state of the lock transition in Doors 2 (locking.locked) is different from Doors 2 (locked) no changes are made to transitions in the edit script. In both Doors 2 and Doors 3 the transition targets the same locked state $d_7$. Figure 3 shows the semantic deltas computed by tmdiff for three consecutive versions of the example state machines. Now the question is, how can they be used to synchronize inter-related artifacts? To make our goals more concrete, let us consider the situation that the state machines are transformed to a Java model to execute them. We assume this transformation propagates the semantic identities of the states in the state machines to the Java code (for instance, using some form of origin tracking [9]). Based on this propagation of identities, it becomes possible to map deltas in terms of state machines to corresponding deltas in terms of Java code. Figure 4 shows hypothetical Java code for Doors 1, and an example of expressing the revision from Doors 1 to Doors 2 in terms of a Java meta model.
Demo: State machines

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

  state opened d3
    close => closed u2

end
```

Screenshot
Future directions

• Time travel (undo, replay)
• Time branching (what-if scenarios)
• Versioning (operation-based)
• Persistence (EventStores!)
• Collaboration (operational transformation)
Live little languages

• DSLs have been shown to be effective for SE
• Live = continuous feedback during programming
• Want: generic techniques for live DSLs
• Need: foundations and engineering principles
• Semantic deltas promising first step