Cascade: a Meta-Language for Change, Cause and Effect

Language-Parametric Enabling Technology that powers Live Programming

Riemer van Rozen
rozen@cwi.nl
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands

ABSTRACT

Live programming brings code to life with immediate and continuous feedback. To enjoy its benefits, programmers need powerful languages and live programming environments for understanding the effects of coding actions and developing running programs. Unfortunately, the enabling technology that powers these languages is missing. Change, a crucial enabler for explorative coding, omniscient debugging and version control, is a potential solution.

In this position paper, we argue that an explicit representation of change is instrumental for how these languages are built, and that cause-and-effect relationships are vital for more precise feedback. We aim to deliver generic solutions for creating these languages. We introduce Cascade, a meta-language and framework for expressing languages with interface- and feedback-mechanisms that drive live programming. Our preliminary results show that Cascade is a promising approach that simplifies developing language back-ends.

KEYWORDS

live programming, live modeling, metamodeling, domain-specific languages, bidirectional model transformations, model migration, REPL languages, coding actions, cascading changes, transactions

1 INTRODUCTION

Live programming caters to the needs of programmers by providing immediate feedback about the effect of coding actions. Figure 1 illustrates a typical coding cycle [9]. Each iteration, programmers make improvements by performing coding actions, events that result in the construction, modification and deletion of objects over time. To help programmers realize their intentions, live programming environments offer suitable user interface mechanisms that enable performing the effects of coding actions. In addition, these environments offer feedback mechanisms that display changes for perceiving those effects, and evaluating if the action has been successful. Good feedback supports forming mental models and learning cause-and-effect relationships that help programmers predict effects of coding actions and make targeted improvements.

Despite the compelling advantages of live programming, its adoption remains sporadic due to a lack of enabling technology for creating the necessary programming languages. Unfortunately, creating languages whose users enjoy the advantages of live programming

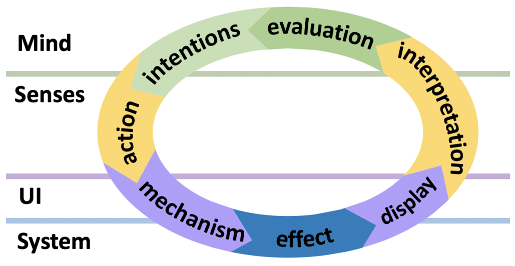


Figure 1: Live Programming speeds-up coding cycles

is incredibly complex, time-consuming and error-prone. Language engineers lack reusable abstractions and techniques to account for run-time scenarios with eventualities such as run-time state migrations, e.g., removing the current state of a state machine.

Several Domain-Specific Languages (DSLs) support a form of live programming that modifies running programs, e.g., Machinations [15], and the State Machine [16] and Questionnaire [14] languages. However, these are one-off solutions with hand-crafted interpreters that are difficult to extend and maintain.

We propose studying how to create these languages in a principled manner, how to express their liveness, and how to add this liveness to existing ones. We argue that an explicit representation of change, a crucial enabler for explorative coding, omniscient debugging and version control, is the missing factor in the currently available language technology. Our main objective is to deliver generic reusable solutions for creating change-driven languages, which foreground cause-and-effect relationships, and let programmers perceive the effects of coding actions.

We propose a novel meta-modeling approach that addresses this challenge. Cascade is a meta-language for prototyping languages whose users enjoy the benefits of live programming. Bi-directional model transformations enable expressing the cascading changes and trickle effects central to live programming. Cascade's interpreter Delta alleviates the burden of prototyping live programming environments. When Delta executes events, it generates run-time scenarios and migrations as cause-and-effect chains. These transactions update a live program's syntax and run-time state.

We introduce preliminary work on Cascade in Section 2. We perform a limited case study in Section 3. We discuss costs and benefits, and describe related work in Sections 4 and 5. Our preliminary results show Cascade is a promising approach that simplifies creating back-ends of change-driven languages, e.g., "live DSLs".

Table 1: The REPL language enables using live languages

function	description	command
do	perform a coding action or user action	call event
undo	roll back a transaction	undo
perceive	inspect state	print
play	replay a history cause-effect-chains	future feature
rewind	unroll a history of cause-effect-chains	future feature
view effect	display a cause-effect-chain textually	(output)
find cause	inspect a textual cause-and-effect-chain	(output)

2 CASCADE

Cascade is a meta-language and a framework for prototyping languages whose users enjoy the benefits of live programming[1]. Cascade offers a generic feedback mechanism that foregrounds cause-and-effect. Cascade has key features for developing language back-ends whose events can be used to provide the feedback programmers need. Language engineers can express *cascading changes* that are central to designing run-time state migrations and trickle effects of live programming. Cascade's interpreter Delta offers a built-in Read-Execute-Print-Loop (REPL), illustrated by Table 1, that runs in the browser as a default interface for textual interaction and testing. We introduce Cascade's features by discussing a running example the Live State Machine Language (LiveSML), a benchmark DSL for live programming of running state machines.

2.1 Language Engineering Questions

Expressing live modeling languages that permit many run-time scenarios with run-time state migrations is intrinsically complex. Therefore, reasoning about *eventualities* is essential. However, language engineers lack reusable abstractions that can account for every eventuality. Linear code execution cannot easily express events that may or may not need to happen before or after every other event. As a result, expressing run-time scenarios is difficult, time-consuming and error-prone. A language engineer who creates a language back-end faces the following technical challenges during its design and construction, e.g., how to:

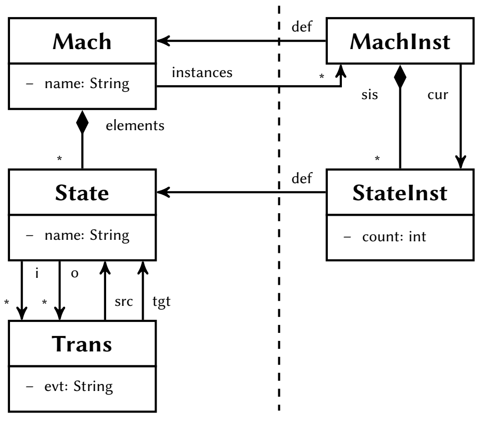
(1) express its abstract syntax and run-time with meta-models.
(2) design the effects of a) *coding actions* that work on its abstract syntax and b) *user interactions* that affect the run-time state.
(3) leverage the REPL features of Table 1 that provide mechanisms for programming, user interaction and feedback.
(4) express cascading changes such as *run-time state migrations* that account for eventualities in run-time scenarios.

We introduce Cascade's features using a running example of LiveSML. Sections 2.2, 2.3 and 2.4 respectively answer Questions 1, 2 and 3. Sections 2.5 and 2.6 partially answer Question 4. In Section 3, we demonstrate a more elaborate migration scenario.

2.2 Models and meta-models

Cascade expresses languages and changes using meta-models. Programs are models that conform to the meta-model of the language. In particular, these models are Abstract Syntax Graphs (ASGs) whose semantics steer the program's behavior. The run-time state is also a model, storing the results of user actions.

[1]https://github.com/vrozen/Cascade



(a) Static metamodel **(b) Run-time metamodel**

Figure 2: Static and run-time metamodels of LiveSML

```
class Mach {
  String name;
  Set<State> states;
  Set<MachInst> instances;
  ...
```

(a) Machine

```
class State {
  String name;
  Set<Trans> i;
  Set<Trans> o;
  ...
```

(b) State

```
class Trans {
  ref State src;
  String evt;
  ref State tgt;
  ...
```

(c) Transition

```
class MachInst {
  ref Mach def;
  ref StateInst cur;
  Map<State, StateInst> sis;
  ...
```

(d) Machine Instance

```
class StateInst {
  ref State def;
  int count;

  ...
```

(e) Machine Instance

Figure 3: Cascade definitions of LiveSML's metamodels

For instance, Figure 2 shows the UML class diagram of LiveSML's meta-model. The static meta-model (on the left), defines the abstract syntax. A machine consists of a number of states with transitions between them. The run-time meta-model (on the right) expresses *running state machines.* A running machine has a current state, and register how often it has resided in each state.

Cascade offers a notation for meta-models that resembles object-oriented programming, shown in Figure 3. Aside from classes, it supports the base types `String`, **`int`**, **`bool`**, and **`enum`**, and the composite types `List`, `Set` and `Map`. Attribute ownership is explicit, and by default, a class owns its attributes. The **`ref`** keyword denotes an alias. We omit visibility because the aim is encapsulating change.

2.3 Coding actions and transformations

Cascade, as a back-end language, does not distinguish between coding actions and user interactions. Mechanisms for both can be expressed using three kinds of parameterized event declarations, called **`effect`**, **`trigger`** and **`signal`**. An *effect* describes how a specific object (referred to as the subject) is created, modified or deleted by means of a bi-directional model transformation. A *trigger* is an input event that has no direct effect, but can trigger other events, side-effects that happen afterwards. A *signal* is an output event that flags an occurrence such as an exception or an error.

```
 1 effect Create(future Mach m, String name) {
 2   m = new Mach();
 3   m.name = name;
 4   m.states = new Set<State>();
 5   m.instances = new Set<MachInst>();
 6 }
 7
 8 inverse effect Delete(past Mach m, String name = m.name) {
 9   m.name = null;
10   delete m.states;
11   delete m.instances;
12   delete m;
13 } pre {
14   foreach(State s in m.states) { State.Delete(s, s.name, m); }
15   foreach(MachInst mi in m.instances) { MachInst.Delete(mi, m); }
16 }
```

Figure 4: LiveSML Mach.Create and Mach.Delete in Cascade

```
LiveSML.Mach.Create(|uuid://5|, "door") {
  [|uuid://5|] = new Mach();
  [|uuid://5|].name = null → "door";
  [|uuid://6|] = new Set<State>();
  [|uuid://5|].states = null → [|uuid://6|];
  [|uuid://7|] = new Set<MachInst>();
  [|uuid://5|].instances = null → [|uuid://7|];
}
```

Figure 5: Transaction resulting from calling Mach.Create

Instead of operating on objects directly, these events operate on object *life-lines*. Parameters and variables can refer to *issued* objects that will exist but do not yet exist, *bound* objects that do exist, and *retired* objects that no longer exist.

Figure 4 shows two effects that work on State Machine definitions, or Mach objects. The Create effect produces a **future** `Mach m` with a specified name (line 1). When called, its operands must be an issued Mach object and a constant string value. Its body (lines 2–5) consists of edit operations, bi-directional atomic instructions executed by Cascade's interpreter Delta . These operations create the Mach object (line 2), set its name (line 3), and create a new sets of states (line 4) and machine instances (line 5).

Passing **future** operands enables scheduling events and side-effects in a non-linear manner. In Section 3, we show how this feature helps account for eventualities.

2.4 Transactions, commits and histories

Language engineers can use Cascade's compiler to obtain a language that integrates with its interpreter Delta. Delta includes a built-in REPL, a textual interface for performing coding actions and user actions, and for perceiving the effects of those actions. For instance, we can create a new state machine called "door" by calling the Create effect shown in Figure 4 from the REPL as follows. Note the ↩ symbol indicates REPL input (pressing the return key), and its absence indicates output, feedback the system gives in response.

```
var m; ↩
LiveSML.Mach.Create(m, "door"); ↩
```

When executed, an event becomes a *transaction* that binds the variables of edit operations to historical constants, e.g., replaced values, implicit indexes, and deleted types. Each transaction thus consists of *bound* edit operations, reversible atomic rewrite rules that work on objects, lists, sets and maps [1, 16]. Each operation has an inverse with the opposite effect. As a result, each transaction can be undone by running the inverse of the sequence in reverse.

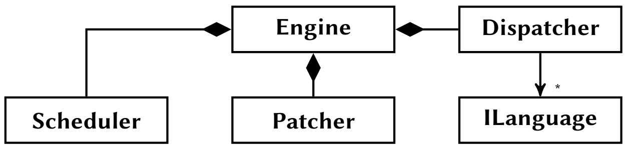


Figure 6: Overview of Delta's extensible engine

Creating a new machine produces the transaction shown in Figure 5, where each variable is bound to a constant. Objects have Unique Universal Identifiers (UUIDs). Inside brackets these indicate object lookups. Every assignment becomes a replacement that replaces old values by new ones. We omit the transactions produced by the REPL language itself, which also saves `m` in its variable store. We verify the results by reading the output from the REPL.

```
m; ↩
machine door
```

Each transaction is part of a *history of commits*, just like in version control systems. In fact, Cascade's interpreter Delta shares many of the characteristics of those systems, but instead of storing textual differences, it stores model differences. Model differences are not merely a textual storage format for change, but also run-time entities interpreted by Delta. Next, we explain how events with side-effects can also produce entire cause-and-effect chains.

2.5 Cascading changes and side-effects

Effects can have side-effects specified in their **pre** and **post** clauses, which when scheduled, happen before and after its edit operations.

For instance, `Mach.Delete`, shown in Figure 4, has a **pre** clause. `Delete` matches `Create`'s parameters, but instead deletes a **past** `Mach m` with its current name, signalling its retirement. The body of edit operations replaces its name by **null** (line 9), deletes the sets of states and instances (lines 10–11), and finally `m` itself (line 12).

However, the sets of states and instances might not be empty when we delete an object. In order to produce a reversible effect, we first have to delete each `State s` from `m.states` (line 14) and also delete each `MachInst mi` from `m.instances` (line 15).

Because in LiveSML there is more than one way in which deletion of these objects can occur, we do not remove each object from its respective set here, and instead leave this responsibility to the respective classes. As a result `Create` and `Delete` are bi-directional transformations that produce opposite effects.

2.6 Local consistency

Cascade adds the notion of *local consistency*. The local validity, or shelf-life, of objects is explicit due to the **future** and **past** keywords of event parameters. Due to these guarantees, language engineers can schedule, or plan, how effects and migrations will be composed from events at run time. Because events are composable, language engineers can reuse model-transformations, and migration trajectories, as side-effects that happen before or after an event. In Section 3 we demonstrate how Cascade handles a non-linear scenario that result from cascading changes with global effects.

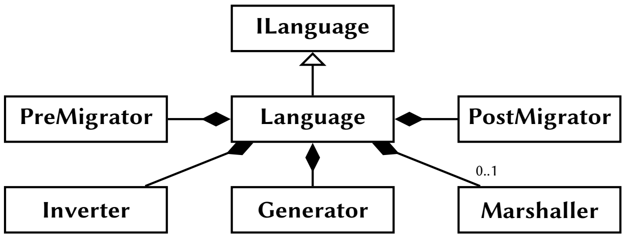


Figure 7: A language consists of a generator and migrators

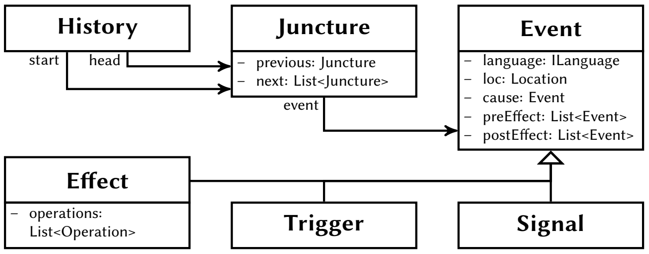


Figure 8: Delta's history consists of cause-and-effect chains

```
1  void schedule(Patcher p, Dispatcher d, Event e) {
2    d.resolve(e);
3    foreach(Event pre in d.preMigrate(e)) { schedule(pre); }
4    d.generate(e);
5    p.commit(e);
6    foreach(Event post in d.postMigrate(e)) { schedule(post); }
7  }
```

Figure 9: C# pseudo-code of Delta's event scheduler

2.7 Cascade framework

The framework consists of a compiler written in Rascal [6], and Delta, an interpreter written in C#. The compiler translates Cascade specifications into language modules that integrate with Delta's extensible engine, illustrated by Figure 6.

Each Cascade package produces three sub-packages: 1) Model contains the classes of the meta-model; 2) Operation contains the classes representing events; and 3) Runtime contains components used to process events and transform models, illustrated by Figure 7.

The engine has three main components. The dispatcher manages a set of languages, and determines which language handles an event. The patcher executes edit operations. The scheduler determines the order in which events are scheduled, generated and migrated. Next, we explain how non-linear event scheduling works.

2.7.1 Scheduling Events. The engine produces transactions in the form of cause-and-effect chains, as illustrated by Figure 8. When called, the scheduler binds an event to a specific object. We sketch the recursive algorithm that schedules each event in Figure 9.

First, the dispatcher resolves the language that processes the event (line 2). Before processing the event itself, the *pre-migrator* of the language determines if any events need to happen before, and if so, those are scheduled first (line 3). Only when the recursive *pre-side-effects* have completed, the generator of the language generates the edit operations that perform the event's own effect (line 4). The patcher immediately commits the transaction to the history, before the operations go stale (line 5). Afterwards, post-migration determines if any events need to happen afterwards (lines 6). When each of those events completes, the event itself completes.

2.7.2 Implementation. The compiler consists of 3285 LOC of Rascal. Delta consists of 11.5 KLOC of C#: edit operations model (1745 LOC), runtime (7288 LOC), and REPL language (2603 LOC).

3 CASE STUDY

The validation of the Cascade framework is ongoing work. As a start, we reproduce a particular live programming scenario on LiveSML. This scenario, previously described by van Rozen and van der Storm [16], and Tikhonova et. al. [13], involves a non-trivial run-time state migration. The preliminary results show that Cascade is a promising approach to create the back-end of change-driven languages, in particular, DSLs.

3.1 Setup

We implement LiveSML in Cascade and we use Delta's REPL to reproduce the following live programming scenario. In this experiment, there are two roles: a programmer who performs coding actions, and a user, who performs user interactions.

The programmer creates a state machine called *doors*, which has three states: *opened*, *closed* and *locked*. The transitions between states offer the user the opportunity to affect the run-time state. In particular, the user can *open* a closed door or *lock* it, *unlock* a locked door, and *close* an open door.

The user creates a running state machine. Its initial state is closed. The user then locks the door. Finally, the programmer remove the locked state from the source code. The expected result is a run-time state migration that resets the current state to the first state in the program: closed, and increases its count by one.

3.2 Results

The Cascade implementation of LiveSML appearing in Appendix A counts 213 LOC (2210 LOC of generated C#). We explain the actions of the programmer the user step by step. The results are as follows.

3.2.1 Creating initial doors program sources. Instead of parsing a state machine program, Delta supports sequential coding actions that work directly on the program's Abstract Syntax Graph (ASG). The programmer creates a new machine *doors* that contains a *closed* state. Using the `print` command, the programmer calls the pretty printer, and verifies the textual program is as expected.

```
LiveSML.Mach.Create(m, "door"); ↩
LiveSML.State.Create(s1, "closed", m); ↩
print m; ↩
machine door
  closed
```

3.2.2 Running a live program instance. Although the program is not yet complete, users can already run it. In live programming, coding- and user- actions can alternate and happen at any time. The user runs the program by creating an instance. Delta does not include a visual user interface with automatic rendering. Instead, the user calls `print` to observe initially its current state (*) is closed.

```
LiveSML.MachInst.Create(mi, m); ↩
print mi; ↩
door
  closed : 1 *
```

3.2.3 *Completing the doors program's sources.*

The programmer now completes the program by adding *opened* and *locked* states, and the transitions between those states.

```
LiveSML.State.Create(s2, "opened", m); ⮐
LiveSML.State.Create(s3, "locked", m); ⮐
LiveSML.Trans.Create(t1, s1, "open", s2); ⮐
LiveSML.Trans.Create(t2, s2, "close", s1); ⮐
LiveSML.Trans.Create(t3, s1, "lock", s3); ⮐
LiveSML.Trans.Create(t4, s3, "unlock", s1); ⮐
```

The programmer verifies the program is complete.

```
print m; ⮐
machine door
  closed
    "open" -> opened
    "lock" -> locked
  opened
    "close" -> closed
  locked
    "unlock" -> closed
```

The user observes the state of the running instance has also been updated. The text between brackets denote "buttons" for the available actions. The user can now *open* or *lock* the closed door.

```
print mi; ⮐
door
  [open] [lock]
  closed : 1 *
  opened : 0
  locked : 0
```

3.2.4 *Locking the door.*

The user locks the door as follows.

```
LiveSML.MachInst.Trigger(mi, "lock"); ⮐
print mi; ⮐
door
  [unlock]
  closed : 1
  opened : 0
  locked : 1 *
```

3.2.5 *Removing the locked state.*

Finally, the programmer removes the locked state. As expected, this causes a run-time state migration, setting the current state to closed, and increasing its count by one.

```
LiveSML.State.Delete(s3, "locked", m); ⮐
print mi; ⮐
door
  [open]
  closed : 2 *
  opened : 0
```

The migration is a cause-effect-chain of transactions that encode the run-time scenario. Its textual representation consists of historical events that are represented by blocks (indicated by braces). Blocks are preceded by the event signature showing its name and its bound operands (between the parentheses). Inside these blocks are nested events and edit operations that have happened in sequence. In the textual representation below, which omits operations, we can observe side-effects. Appendix B shows a complete trace.

```
State.Delete(uuid26, "locked", uuid13) {
  Trans.Delete(uuid33, uuid26, "unlock", uuid16) {
    State.RemoveOut(uuid26, uuid33) { /*operations*/ }
    State.RemoveIn(uuid16, uuid33) { /*operations*/ }
    /*operations*/ }
  Trans.Delete(uuid32, uuid16, "lock", uuid26) {
    State.RemoveOut(uuid16, uuid32) { /*operations*/ }
    State.RemoveIn(uuid26, uuid32) { /*operations*/ }
    /*operations*/ }
  Mach.RemoveState(uuid13, uuid26) {
    MachInst.RemoveStateInst(uuid19, uuid29, uuid26) {
      MachInst.SetCurState(uuid19, null, uuid29) { /*operations*/ }
      /*operations*/ }
    StateInst.Delete(uuid29, uuid26, uuid26) { /*operations*/ }
    MachInst.Initialize(uuid19) {
      MachInst.SetCurState(uuid19, uuid21, null) {
        /*operations*/
        StateInst.SetCount(uuid21, 2, 1) { /*operations*/ } } }
    /*operations*/ }
  /*operations*/ }
```

Deleting the locked state has caused several side-effect. First, the transitions lock and unlock were deleted. Next, the locked state were removed from the machine, which also had side-effects. The state instance was removed from the machine instance, which also set the current state of the machine instance to null. The state instance was then deleted. Finally, the machine instance reinitialized, which set its current state to locked and raised the count. Note that the operations that delete the locked state actually happen last.

3.3 Analysis

The case of LiveSML demonstrates that Cascade can express DSLs for live programming. The implementation of LiveSML in Cascade is relatively small (213 LOC), but does account for many non-trivial migration scenarios. However, as the results show, these scenarios quickly grow. Analyzing lengthy sequences of side-effects is a non-trivial task. Instead of the textual representation on the REPL, language designers and programmers need an interactive debugger that supports root-cause analysis, e.g., via a filtering mechanism. Because cause-effect-chains are annotated with source locations, we can relate them to their Cascade sources for debugging via origin tracking. Of course, the case of LiveSML is not a full validation of our framework. We are currently conducting additional case studies on Machinations, the Questionnaire Language and Behavior Trees.

4 DISCUSSION

Cascade has compelling benefits. Concise specifications can account for many migration scenarios. Delta generates a history at no additional cost. This history contains every transaction of every change that took place, and why it happened. That facilitates omniscient debugging, rolling back time, and exploring what-if scenarios. Delta not only ensures the run-time state is correct, it also produces cause-and-effect chains that can support explorative coding, omniscient debugging and version control.

Because the generated language back-ends are event-driven, they combine well with visual user interface designs and parser front-ends. Engineers can leverage external libraries and components, e.g., parser generators and browser-based user interfaces.

Of course, there are also costs. For language engineers, bidirectional thinking is not straightforward, and language designs do not normally include run-time state migrations. Learning how this works takes time and effort. At this time, the formal properties of Cascade are not yet known, and its limitations not yet explored.

The following sections discuss additional design considerations.

4.1 Bi-directionality and inverse effects

Bi-directionality of transformations in Cascade comes at the cost of garbage collection. Creating a new object is, as one would expect, a sequence of operations that create new objects and *afterwards* assign initial values. However, deleting an object is more involved. Deletion requires a clean-up of every child object owned, and typically also erasure of references to the object (or aliases) *before* the object can be deleted itself. To ensure bi-directionality, fields must have default values before deletion. Lists, sets and maps must be empty, because when created they start out that way. As a result, well-formed create and `inverse` delete effects provide full store- and load-functionality. The design decision, that all change must be explicit, adds some some verbosity, but ensures events can always be related to their Cascade specification. We observe that default code for events and inverses may be generated, and that explicit ownership of attributes enables its static analysis.

4.2 Marshalling models

In the deployment of model-transformation engines, one needs to consider embedding context in which it will be used. In Delta, instances of meta-models are ordinary C# objects. Marshalling involves first annotating containment trees. Paradoxically, deleting a model stores it in the history, thus marshalling its events. Therefore, restoring the model is as simple as running its deletion in reverse.

4.3 Language composition

Domain-Specific Languages often combine or reuse parts of other languages. With its side-effects, Cascade offers a natural extension point for combining languages. Language modules can be reused using the import statement. This feature enables calling events from the built-in REPL language, e.g., the events of LiveSML.

4.4 Fixpoint computations

Live programming often takes the form of a trickle effect whose ripples move through the system until no more effects can be observed. These effects can be expressed using *fixpoint computations*.

Because language engineers can use Cascade to schedule side-effects before and after each event happens, they can schedule future checks that determine if the process is done. Both Machinations and the Live Questionnaire Language contain fixpoints.

5 RELATED WORK

Live Programming is a research area that intersects Programming Languages (PL) and Human-Computer Interaction (HCI). The term refers to a wide array of user interface mechanisms, language features and debugging techniques that revolve around iterative changes and immediate and continuous feedback [10].

Tanimoto describes *levels of liveness* that help distinguish between forms of feedback in live programming environments [12]. Each level adds a property: 1) informative or descriptive; 2) executable; 3) responsive or edit triggered; and 4) live or stream driven.

Many forms of live programming exist, each designed with different goals in mind. For instance, McDirmid describes how *probes*, a mechanism interwoven in the editor, helps diagnosing problems [8]. Ko describes *whyline*, a debugging mechanism for asking why-questions about Java program behavior [7]. Another angle is extending interpreters with a so-called Read-Eval-Print-Loop (REPL) [2], a textual interface for executing commands sequentially. Omniscient debugging supports easily navigating between program states, and exploring what-if scenarios, during a program's execution [3].

Several Domain-Specific Languages (DSLs) offer live programming, e.g., Machinations [15], and the State Machine [16] and Questionnaire [14] languages. Machinations is a DSL for game economies that lets game designs modify the rules of running game software, potentially adding and removing playful affordances. The Questionnaire Language (QL) is a DSL for digital forms originally intended for tax office. QL enables modifying the sources of partially answered forms with dependent sections, which can cause migrations and update visibility.

The area of *modelware* is a technological space that revolves around the design, maintenance and reuse of models. Model transformations are a key technology for expressing change. In their seminal paper on "the difference and union of models", Alanen and Porres describe a notation, originally intended for model versioning, known as edit operations (or edit scripts), which expresses model deltas. Van der Storm proposes creating live programming environments driven by "semantic deltas", based on this notation [14].

Bi-directional Model Transformation (BX) is a well-researched topic that intersects with several areas [5]. BX has impacted relational databases, model-driven software development [11], graphical user interfaces, visualizations with direct manipulation, structure editors, and data serialization, to name a few. Cicchetti et al describe the Janus Transformation Language (JTL), a language for bi-directional change propagation [4].

The study of live modeling with run-time state migrations initially focused on patching edit operations [16]. Constraint-based solutions instead focus on correct states with respect to a set of constraints [13], but not on change. Until now, no solution could explain why a particular migration happened. Cascade is a BX-based solution that addresses both *why* and *how*, and considers migrations as an integral part of the language semantics.

6 CONCLUSION

Programmers need powerful languages and programming environments to enjoy the benefits of live programming. However, the enabling technology that powers these languages is missing. We have addressed this challenge by proposing a novel meta-modeling approach. We have introduced Cascade, a meta-language and framework for prototyping languages and live programming environments. Our preliminary results show that Cascade is a promising approach that simplifies creating back-ends of change-driven languages, e.g., "live DSLs". Future work includes the following.

6.1 Future work

Of course, live programming environments consist editors and visualizers too. The REPL traces are not well suited for human interpretation. Instead, programmers need visual user interfaces that present feedback and focus attention on specific changes for observing and interpreting effect. The shape of a DSL matters for its expressiveness and understandability, e.g. block-based syntax. An open challenge is leveraging generic technology that applies semiotics to display appropriate icons and imagery.

Unlike languages whose AST remains fixed during the a program's run time, languages with live run-time state migrations can be considered *unsound*. We have observed local consistency, but investigating to what extent global formal correctness properties can be guaranteed is an open research challenge.

We have conducted a limited case study on LiveSML. In additional case studies, we will study the liveness qualities of the Questionnaire Language, Machinations and Behavior Trees.

ACKNOWLEDGMENTS

I thank Tijs van der Storm, Daria Polak and the LIVE program committee for their comments that helped improve this paper.

REFERENCES

[1] Marcus Alanen and Ivan Porres. 2003. Difference and Union of Models. In *«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)*. Springer, 2–17. https://doi.org/10.1007/978-3-540-45221-8_2

[2] L. Thomas van Binsbergen, Mauricio Verano Merino, Pierre Jeanjean, Tijs van der Storm, and Olivier Combemale, Benoîtand Barais. 2020. A Principled Approach to REPL Interpreters. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2020, Virtual, November, 2020*. ACM, 84–100. https://doi.org/10.1145/3426428.3426917

[3] Erwan Bousse, Dorian Leroy, Benoît Combemale, Manuel Wimmer, and Benoit Baudry. 2018. Omniscient Debugging for Executable DSLs. *J. Syst. Softw.* 137 (2018), 261–288. https://doi.org/10.1016/j.jss.2017.11.025

[4] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. 2010. JTL: A Bidirectional and Change Propagating Transformation Language. In *Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12–13, 2010, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6563)*. Springer, 183–202. https://doi.org/10.1007/978-3-642-19440-5_11

[5] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. 2009. Bidirectional Transformations: A Cross-Discipline Perspective. In *Theory and Practice of Model Transformations - 2nd International Conference, ICMT@TOOLS 2009, Zurich, Switzerland, June 29–30, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5563)*. Springer, 260–283. https://doi.org/10.1007/978-3-642-02408-5_19

[6] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20–21, 2009*. IEEE Computer Society, 168–177. https://doi.org/10.1109/SCAM.2009.28

[7] Amy J. Ko and Brad A. Myers. 2004. Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior. In *Proceedings of the 2004 Conference on Human Factors in Computing Systems, CHI 2004, Vienna, Austria, April 24 - 29, 2004*. ACM, 151–158. https://doi.org/10.1145/985692.985712

[8] Sean McDirmid. 2013. Usable Live Programming. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*. ACM, 53–62. https://doi.org/10.1145/2509578.2509585

[9] Donald A. Norman. 1986. Cognitive Engineering. *User centered system design* 31 (1986), 61.

[10] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2019. Exploratory and Live, Programming and Coding - A Literature Study Comparing Perspectives on Liveness. *Art Sci. Eng. Program.* 3, 1 (2019), 1. https://doi.org/10.22152/programming-journal.org/2019/3/1

[11] Perdita Stevens. 2010. Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. *Softw. Syst. Model.* 9, 1 (2010), 7–20. https://doi.org/10.1007/s10270-008-0109-9

[12] Steven L. Tanimoto. 2013. A Perspective on the Evolution of Live Programming. In *Proceedings of the 1st International Workshop on Live Programming, LIVE 2013, San Francisco, California, USA, May 19, 2013*. IEEE Computer Society, 31–34. https://doi.org/10.1109/LIVE.2013.6617346

[13] Ulyana Tikhonova, Jouke Stoel, Tijs van der Storm, and Thomas Degueule. 2018. Constraint-based Run-time State Migration for Live Modeling. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018*. ACM, 108–120. https://doi.org/10.1145/3276604.3276611

[14] Tijs van der Storm. 2013. Semantic Deltas for Live DSL Environments. In *Proceedings of the 1st International Workshop on Live Programming, LIVE 2013, San Francisco, California, USA, May 19, 2013*. IEEE Computer Society, 35–38. https://doi.org/10.1109/LIVE.2013.6617347

[15] Riemer van Rozen and Joris Dormans. 2014. Adapting Game Mechanics with Micro-Machinations. In *Proceedings of the 9th International Conference on the Foundationsof Digital Games, FDG 2014, Liberty of the Seas, Caribbean, April 3–7, 2014*. Society for the Advancement of the Science of Digital Games.

[16] Riemer van Rozen and Tijs van der Storm. 2019. Toward Live Domain-Specific Languages - From Text Differencing to Adapting Models at Run Time. *Softw. Syst. Model.* 18, 1 (2019), 195–212. https://doi.org/10.1007/s10270-017-0608-7

A LIVE STATE MACHINE LANGUAGE

This appendix describes the Live State Machine Language (LiveSML). We give a complete implementation of its back-end in the Cascade meta-language. Omitting a visual user interface, but including a pretty-printer (not shown here), LiveSML counts 213 lines of code.

LiveSML has a meta-model that consists of three classes: State, Trans and Mach. Programmers can create, manipulate and delete a program's parts via *coding actions* that work on the syntax tree.

Users can run a machine, and manipulate its run-time state through *user interactions*. Live programs have a run-time meta-model that adds the classes MachInst and StateInst. The run-time state migrates with changes to the source code of the program.

Machine. A machine is a named sequence of states with transitions between them that manages a set of running instances. Programmers can create machines before users can run them. Deleting a machine also deletes its states and any running instance.

Adding and removing states to machines are side-effects. However, in response to these effects machines also and removes state instances to machine instances, and reinitializes them.

```
package Delta.LiveSML {
  class Mach {
    String name;
    Set<State> states;
    Set<MachInst> instances;
    effect Create(future Mach m, String name) {
      m = new Mach();
      m.name = name;
      m.states = new Set<State>();
      m.instances = new Set<MachInst>();
    }
    inverse effect Delete(past Mach m, String name = m.name) {
      m.name = null;
      delete m.states;
      delete m.instances;
      delete m;
    } pre {
      foreach(State s in m.states) { State.Delete(s, s.name, m); }
      foreach(MachInst mi in m.instances) { MachInst.Delete(mi, m); }
    }
    side-effect AddState(Mach m, State s) {
      m.states.add(s);
    } post {
      foreach(MachInst mi in m.instances) {
        begin StateInst si;
        StateInst.Create(si, s);
        MachInst.AddStateInst(mi, si, s);
        MachInst.Initialize(mi);
      }
    }
    inverse side-effect RemoveState(Mach m, State s) {
      m.states.remove(s);
    } pre {
      foreach(MachInst mi in m.instances) {
        StateInst si = mi.sis[s];
        MachInst.RemoveStateInst(mi, si, s);
        StateInst.Delete(si, s);
        MachInst.Initialize(mi);
      }
    }
    side-effect AddMachInst(Mach m, MachInst mi)
      { m.instances.add(mi); }
    inverse side-effect RemoveMachInst(Mach m, MachInst mi)
      { m.instances.remove(mi); }
  } }
```

State. Programmers can create new named states as part of an existing machine. Deleting a state from its machine also deletes its incoming and outgoing transitions, ensuring program consistency.

```
package Delta.LiveSML {
  class State {
    String name;
    Set<Trans> i;
    Set<Trans> o;
    effect Create(future State s, String name, Mach m) {
      s = new State();
      s.i = new Set<Trans>();
      s.o = new Set<Trans>();
      s.name = name;
    } post { Mach.AddState(m, s); }
    inverse effect Delete(past State s, String name = s.name, Mach m) {
      s.name = null;
      delete s.i;
      delete s.o;
      delete s;
    } pre {
      foreach(Trans t in s.o) { Trans.Delete(t, t.src, t.evt, t.tgt); }
      foreach(Trans t in s.i) { Trans.Delete(t, t.src, t.evt, t.tgt); }
      Mach.RemoveState(m, s);
    }
    side-effect AddIn(State s, Trans t) { s.i.add(t);  }
    inverse side-effect RemoveIn(State s, Trans t) { s.i.remove(t); }
    side-effect AddOut(State s, Trans t) { s.o.add(t); }
    inverse side-effect RemoveOut(State s, Trans t) { s.o.remove(t); }
  } }
```

Transition. Transitions determine how a running machine responds to events by moving between states. Programmers can create and delete transitions when the source and target states exist. Transitions add and remove themselves from the respective states.

```
package Delta.LiveSML {
  class Trans {
    ref State src;
    String evt;
    ref State tgt;
    effect Create(future Trans t, State src, String evt, State tgt) {
      t = new Trans();
      t.src = src;
      t.evt = evt;
      t.tgt = tgt;
    } post {
      State.AddOut(src, t);
      State.AddIn(tgt, t);
    }
    inverse effect Delete(past Trans t, State src=t.src, String evt=t.evt, State
        tgt=t.tgt) {
      t.src = null;
      t.evt = null;
      t.tgt = null;
      delete t;
    } pre {
      State.RemoveOut(src, t);
      State.RemoveIn(tgt, t);
    } } }
```

State Instance. A state instance keeps count of how many times a state has been visited in a particular running machine. Creation, deletion, and updates to the count are all fully automatic side-effects.

```
package Delta.LiveSML {
  class StateInst {
    ref State def;
    int count;
    side-effect Create(future StateInst si, State s) {
      si = new StateInst();
      si.count = 0;
      si.def = s;
    }
    inverse side-effect Delete(past StateInst si, State def = si.def) {
      si.def = null;
      si.count = 0;
      delete si;
    }
    invertible side-effect SetCount(StateInst si, int count=si.count) {
      si.count = count;
    } } }
```

Machine Instance. A machine instance represents a running state machine that maps each state to a state instance. Users can create and delete instances, and trigger transitions between state instances. Run-time state migrations that result from changes to its machine definition, keep the map and the current state valid and up to date.

```
package Delta.LiveSML {
  class MachInst {
    ref Mach def;
    ref StateInst cur;
    Map<State, StateInst> sis;
    effect Create(future MachInst mi, Mach m) {
      mi = new MachInst();
      mi.def = m;
      mi.cur = null;
      mi.sis = new Map<State, StateInst>();
    } post {
      Mach.AddMachInst(mi.def, mi);
      foreach(State s in mi.def.states) {
        begin StateInst si;
        StateInst.Create(si, s);
        AddStateInst(mi, si, s);
      }
      Initialize(mi);
    }
    inverse effect Delete(past MachInst mi, Mach def = mi.def) {
      mi.def = null;
      mi.cur = null;
      delete mi.sis;
      delete mi;
    } pre {
      Mach.RemoveMachInst(mi.def, mi);
      foreach(StateInst si in mi.sis.Values) {
        RemoveStateInst(mi, si, si.def); StateInst.Delete(si, si.def);
      }
    }
    side-effect AddStateInst(MachInst mi, StateInst si, State s) {
      mi.sis[s] = si;
    } inverse side-effect RemoveStateInst(MachInst mi, StateInst si, State s) {
      mi.sis.remove(s);
    } pre { if(si == mi.cur){ SetCurState(mi, null); } }
    invertible side-effect SetCurState(MachInst mi, StateInst cur = mi.cur) {
      mi.cur = cur;
    } post {
      if(mi.cur != null) { StateInst.SetCount(mi.cur, mi.cur.count + 1); }
    }
    trigger Initialize(MachInst mi) {
      if(mi.sis.Count > 0 && mi.cur == null) {
        StateInst nextState = mi.sis.Values.First();
        SetCurState(mi, nextState);
      }
    }
    trigger Trigger(MachInst mi, String evt) {
      if(mi.cur == null) {  MissingCurrentState(mi); return; }
      foreach(Trans t in mi.cur.def.o) {
        if(t.evt == evt) {
          StateInst nextState = mi.sis[t.tgt];
          SetCurState(mi, nextState);
          return;
        }
      }
      Quiescence(mi);
    }
    signal MissingCurrentState(MachInst mi);
    signal Quiescence(MachInst mi);
  }
}
```

B RUN-TIME STATE MIGRATION

This appendix shows the effect of deleting state locked in the experiment described by Section 3. The run-time state migration is a cause-and-effect-chain with the following textual representation.

```
Call("LiveSML.State.Delete(s3, \"locked\", m)") {
  /*post effects*/
  State.Delete(uuid26, "locked", uuid13) {
    /*pre effects*/
    Trans.Delete(uuid33, uuid26, "unlock", uuid16) {
      /*pre effects*/
      State.RemoveOut(uuid26, uuid33) {
        /*operations*/
        [|uuid://28|].remove([|uuid://33|]);
      }
      State.RemoveIn(uuid16, uuid33) {
        /*operations*/
        [|uuid://17|].remove([|uuid://33|])
      }
      /*operations*/
      [|uuid://33|].src = [|uuid://uuid26|] ⟶ null;
      [|uuid://33|].evt = "unlock" ⟶ null;
      [|uuid://33|].tgt = [|uuid://uuid16|] ⟶ null;
      delete Trans [|uuid://33|];
    }
    Trans.Delete(uuid32, uuid16, "lock", uuid26) {
      /*pre effects*/
      State.RemoveOut(uuid16, uuid32) {
        /*operations*/
        [|uuid://18|].remove([|uuid://32|]);
      }
      State.RemoveIn(uuid26, uuid32) {
        /*operations*/
        [|uuid://27|].remove([|uuid://32|])
      }
      /*operations*/
      [|uuid://32|].src = [|uuid://uuid16|] ⟶ null;
      [|uuid://32|].evt = "lock" ⟶ null;
      [|uuid://32|].tgt = [|uuid://uuid26|] ⟶ null;
      delete Trans [|uuid://32|];
    }
    Mach.RemoveState(uuid13, uuid26) {
      /*pre effects*/
      MachInst.RemoveStateInst(uuid19, uuid29, uuid26) {
        /*pre effects*/
        MachInst.SetCurState(uuid19, null, uuid29) {
          /*operations*/
          [|uuid://19|].cur = [|uuid://uuid29|] ⟶ null;
        }
        /*operations*/
        [|uuid://20|][[|uuid://26|]] = [|uuid://29|] ⟶ null;
        [|uuid://20|].remove([|uuid://26|]);
      }
      StateInst.Delete(uuid29, uuid26) {
        /*operations*/
        [|uuid://29|].def = [|uuid://26|] ⟶ null;
        [|uuid://29|].count = 1 ⟶ 0;
        delete StateInst [|uuid://29|];
      }
      MachInst.Initialize(uuid19) {
        /*post effects*/
        MachInst.SetCurState(uuid19, uuid21) {
          /*operations*/
          [|uuid://19|].cur = [|uuid://21|];
          /*post effects*/
          StateInst.SetCount(uuid21, 2, 1) {
            //operations
            [|uuid://21|].count = 1 ⟶ 2;
          }
        }
      }
      /*operations*/
      [|uuid://14|].remove([|uuid://26|]);
    }
    /*operations*/
    [|uuid://26|].name = "locked" ⟶ null;
    delete Set<Trans> [|uuid://27|];
    delete Set<Trans> [|uuid://28|];
    delete State [|uuid://26|];
  }
}
```