Cascade: A Meta-Language for Change, Cause & Effect

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 code modifications on 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.

We aim to deliver generic solutions for creating these languages, in particular Domain-Specific Languages (DSLs). We present Cascade, a meta-language for expressing DSLs with interface- and feedback-mechanisms that drive live programming. We demonstrate run-time migrations, ripple effects and live desugaring of three existing DSLs. Our results show that an explicit representation of change is instrumental for how these languages are built, and that cause-and-effect relationships are vital for delivering precise feedback.

CCS Concepts: • **Software and its engineering** → *Visual languages*; *Domain specific languages*; *Integrated and visual development environments*; *Interpreters.*

Keywords: live programming, metamodels, domain-specific languages, bidirectional transformations, model migration

ACM Reference Format:
Riemer van Rozen. 2023. Cascade: A Meta-Language for Change, Cause & Effect. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3623476.3623515

1 Introduction

Live programming caters to the needs of programmers by providing immediate feedback about the effect of changes to the code. Figure 1 illustrates a typical coding cycle [13].

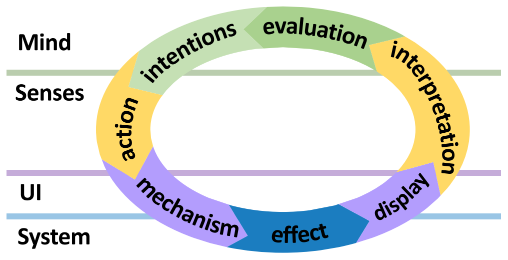


Figure 1. Live Programming speeds up coding cycles

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 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., the State Machine Language (SML) [24], Questionnaire Language (QL) [20] and Machinations [23]. However, these are one-off solutions with hand-crafted interpreters that are difficult to extend and maintain.

We study how to create such DSLs in a principled manner, how to express their liveness, and how to add this liveness to existing ones. We hypothesize that an explicit representation of change, a crucial enabler for exploratory coding, omniscient debugging and version control, is the missing factor in the currently available language technology. Our main objective is to deliver language-parametric solutions for creating change-driven DSLs that foreground cause-and-effect relationships, and let programmers perceive effects.

SLE '23, October 23–24, 2023, Cascais, Portugal
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0396-6/23/10.
https://doi.org/10.1145/3623476.3623515

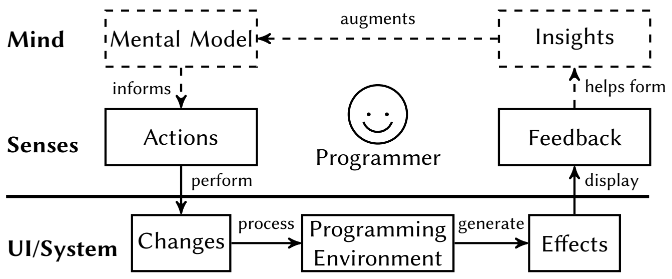
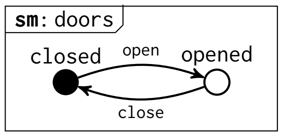


Figure 2. Relating actions to events, feedback and insights

```
machine doors
  state closed
    open => opened
  state opened
    close => closed
```

(a) Textual doors program



(b) Visual program

Figure 3. State Machine Language: doors example

We propose a novel meta-modeling approach that leverages an explicit representation of change. We present Cascade, a meta-language for expressing DSLs with input and feedback mechanisms that drive live programming. Cascade expresses "cascading changes" that introduce liveness using bi-directional model transformations with side-effects.

Using its compiler, language engineers generate interpreters that integrate with Delta, Cascade's runtime. Delta offers a built-in Read-Eval-Print-Loop (REPL) for simulating live programming scenarios that bring the code to life. By executing sequential commands on the REPL, engineers can simulate coding actions, user interaction and feedback.

When Delta executes events, it generates transactions as cause-and-effect chains. These transactions update a live program's syntax and run-time state. We investigate how Cascade can help express the interpreters of SML, QL and Machinations. Our results show Cascade is instrumental for rapidly creating executable DSL prototypes with concise and maintainable designs. Our contributions are: 1) Cascade: a meta-language for change, cause and effect; 2) Delta: a runtime for creating live programming environments; and 3) three case studies that reproduce liveness[1].

2 Problem Overview

We study a form of live programming that works on running programs. We relate the needs of programmers, illustrated by Figure 2, to changes and feedback in Section 2.2. We formulate hypotheses and objectives in Sections 2.3 and 2.4. First, we introduce a scenario that motivates this work.

2.1 Scenario: Modifying a running machine

The Live State Machine Language (LiveSML) is a DSL for simultaneously creating and running state machines.

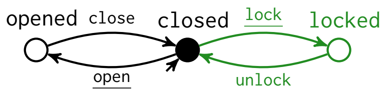
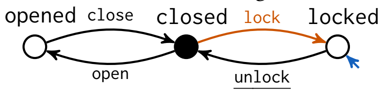
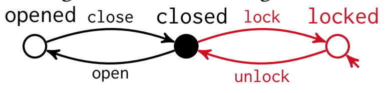
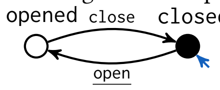
#	Origin	Effect	Run-time state
1.	Coding actions.	Adding state locked. 	closed: 1 * opened: 0 $\boxed{\texttt{locked: 0}}$
2.	User interaction.	Locking the door. 	closed: 1 opened: 0 locked: $\boxed{\texttt{1 *}}$
3.	Coding action.	Deleting the locked state. 	closed: 1 opened: 0 ~~locked: 1 *~~
4.	State migration.	Updating the current state. 	closed: $\boxed{\texttt{2 *}}$ opened: 0

Figure 4. Live programming scenario of a running doors program that demonstrates run-time state migration

We use this DSL as a illustrative example because it is easy to comprehend, and also appears in related work [19, 24]. Figure 3 shows an SML program called doors. When executed, it can be either in the opened or closed state.

As a concise example, Figure 4 describes a live programming scenario of a running doors program. After starting the program of Figure 3, each step shows the origin of a change and the effects on the program and its run-time state.

First, the programmer adds a locked state, and two transitions for locking and unlocking the door. These additions are marked in green. In response, the interpreter introduces a locked count of zero (shown in a box). Initially, the current state (marked *), is closed, and the lock and open transitions (underlined) can be activated. In step two, the programmer triggers the lock transition in the user interface. The interpreter performs the transition (feedback shown in orange), updates the current state to locked, and raises its count (shown in a box). Updates are shown in blue.

Finally, in step three, the programmer deletes the locked state. In response, the interpreter also deletes the lock and unlock transitions (shown in red). Because the current state is removed (indicated with strikethrough), this state has become invalid. In step four, which follows immediately, the interpreter migrates the program to the initial state closed.

2.2 The need for Live Programming

Though limited in its complexity, the scenario illustrates key requirements that programming environments must fulfil to cater to the programmers' needs. Every coding cycle, the challenge is relating feedback about the effects of changes to insights about improvements, as illustrated by Figure 2.

[1]An earlier version of this paper has been presented as: R. van Rozen. 2022. Cascade: A Meta-Language for Change, Cause and Effect: Enabling Technology for Live Programming. In *Workshop on Live Programming, LIVE 2022.*

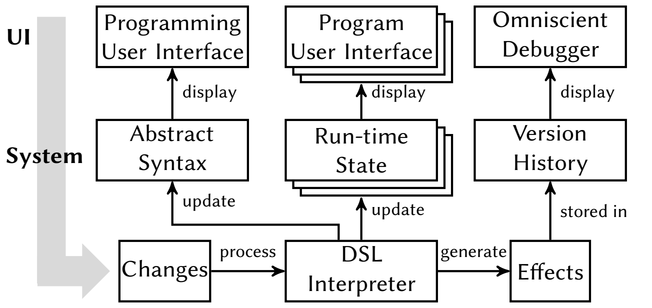


Figure 5. Change-based Live Programming Environment

R1. Gradual change. As software evolves, programmers constantly need to *make changes* to the code. Each iteration, they realize intentions, improve behaviors and fix bugs.

R2. Immediate feedback. Feedback is essential for testing hypotheses and verifying behaviors. For making timely improvements, programmers need feedback with every change.

R3. Evolving behavior. Programmers run programs to evaluate behaviors. For assessing the impact of changes, they need to observe the difference in behavior.

R4. Learnable behavior. For making targeted improvements, programmers need to learn from successes and mistakes. Programmers have to learn to predict the outcomes of changes.

R5. Exploratory design. To support gradually improving insights, programmers need to freely explore design spaces through do, undo, redo, record, and playback functionality.

2.3 Change-based DSL environments

Using programming environments, programmers can perform *coding actions*, events that result in the construction, modification and deletion of program elements. Invalid or syntactically incorrect edits are not coding actions. We study the effects of coding actions, specifically, changes to visual programs that work directly on the abstract syntax and affect running programs. Two main hypotheses drive this study:

1. Live programming can make code come alive in the imagination of the programmer by keeping "test cases" running.
2. DSLs are especially suitable to support live programming and to deliver feedback that appeals to the imagination.

We aim to empower programmers with programming environments for exploring the run-time effects of coding actions.

2.4 Language-parametric enabling technology

We study how to create such DSLs in a principled manner, how to express their liveness, and how to add this liveness to existing ones. We envision change-based live programming environments, as illustrated by Figure 5, based on a set of reusable principles, formalisms and components. For providing live feedback, language designs must account for run-time eventualities, valid changes to programs and run-time states. These cannot easily all be linearly represented due to the multitude of valid executions and dependencies.

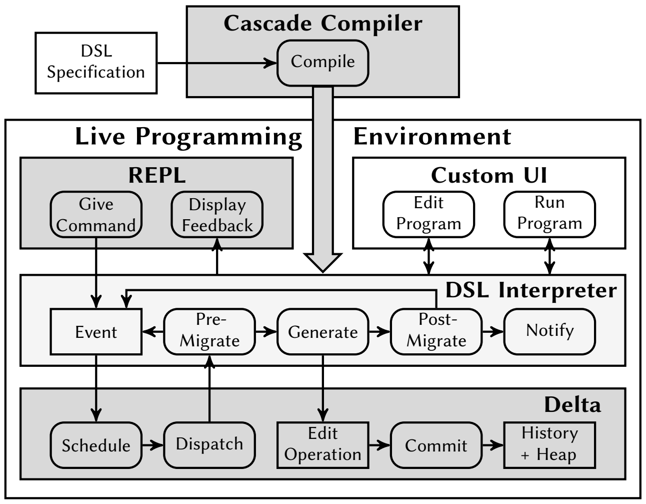


Figure 6. Cascade Framework: Generating DSL Interpreters

We address the need for enabling technology that powers these environments. We aim to simplify language design by abstracting from individual scenarios and the ordering of events. Next, we propose a generic approach that expresses change as modular and reusable model transformations. This solution can steer global run-time executions through local and conditional side-effects defined on the meta-level.

3 Cascade Framework

We present the Cascade framework, a language-parametric solution for developing change-based live programming environments. Cascade, illustrated by Figure 6, offers a meta-language and a set of generic reusable components (gray) that integrate domain-specific additions (white) for addressing the following technical challenges.

3.1 Cascade Meta-Language

Cascade is a meta-language for change, cause and effect. We introduce its features in Section 5. To create DSLs realizing the requirements of Section 2.2 language engineers can:

- T1 Express the syntax and run-time states using meta-models (R1, R3). Section 5.1 introduces these concepts.
- T2 Design actions, events and effects as bi-directional model-transformations that support exploratory coding (R5). Section 5.2 explains how to design interactions that gradually change the syntax and program behaviors (R1–3).
- T3 Design side-effects as relationships between events with predicable outcomes to steer behaviors (R4). Section 5.3 explains how to express mutations of program elements.
- T4 Design cascading changes that are central to live programming, e.g, for expressing run-time state migrations (R3, R4) that must account for many run-time eventualities (R5). Section 5.4 discusses design considerations.

In three case studies, we explore how Cascade helps to express the language designs and program execution of DSLs in a principled manner. In Sections 7, 8 and 9 we investigate:

```
o_new(|uuid://1|, "State");
o_set([|uuid://1|], "name", "opened", null);
o_new(|uuid://2|, "List<Trans>");
o_set([|uuid://1|], "out", [|uuid://2|], null);
o_new(|uuid://3|, "Trans");
o_set([|uuid://3|], "src", "opened", null);
o_set(|[uuid://3|], "evt", "close", null);
l_insert([|uuid://2|], 0, [|uuid://3|]);
```

(a) Example edit operations

```
s = new State();
s.name = "opened";
s.out = new List<Trans>();

t = new Trans();
t.src = s;
t.evt = "close";
s.out.push(t);
```

(b) Generic edit script

Figure 7. Contrasting edit operations from edit scripts

a) consistency and run-time state migrations of LiveSML;
b) trickle-effects and fixpoint computations of LiveQL; and
c) live desugaring and visual feedback of Machinations.

3.2 Live Programming Environments

Cascade provides reusable components for developing live programming environments, easing the authorial burden.

3.2.1 User Interface. Programmers need appropriate input and feedback mechanisms for changing code, obtaining feedback, and observing changes. Cascade offers a choice: 1) develop a user-friendly custom UI, based on its event APIs; or 2) use a generic Read-Eval-Print-Loop (REPL), e.g., before creating a custom UI. In this paper, we explore both. We use the REPL to simulate live programming scenarios textually. By executing sequential commands, we can simulate coding actions, user interaction and feedback on a line by line basis.

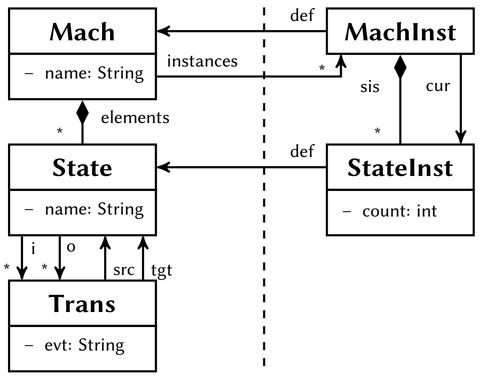
3.2.2 Interpreter. A powerful interpreter is the driving force behind live programming. We formulate technical challenges for creating interpreters that can support the requirements of Section 2.2.

T5 Integrate DSLs in a common run-time environment.
T6 Schedule and execute events that perform bi-directional model transformations and run-time state migrations.
T7 Maintain a version history and a heap, for updating syntax trees and run-time states with gradual changes.
T8 Generate the effects of transformations as historical transactions that: a) capture changes as edit operations; and b) preserve causal relationships in cause-and-effect chains.

To tackle these challenges, Cascade generates a DSL interpreter from its specification. Section 6 discusses the design of the compiler, the generated interpreters and Delta, Cascade's runtime. We begin by introducing edit operations.

4 Edit Operations

Cascade introduces an explicit representation of change for expressing behavioral effects of coding actions, user interactions and program executions. Cascade expresses change as model transformations that work on models (or programs) and run-time states, which each consist of objects. We base its representation on a language variant of the edit operations [2]. Originally introduced for differencing and merging, these operations have since also been used as a low-level storage format for expressing run-time effects [20, 24].



(a) Static meta-model **(b)** Run-time meta-model

Figure 8. Static and run-time meta-models of LiveSML

Cascade leverages edit operations to express transactional effects, maintain a version history, and support exploratory live programming. Figure 7a shows example operations that create new objects and replace attribute values. Appendix A describes a complete set of edit operations that work on commonly used data structures: objects, lists, sets and maps.

However, edit operations alone are not sufficient. An expressive meta-language for change requires variables, not just values. Cascade introduces a script notation, illustrated by Figures 7b, that resolves this issue. Next, we explain how Cascade's transformations encapsulate these edit scripts.

5 The Cascade Meta-Language

Cascade is a meta-programming language for expressing change, cause and effect. Using Cascade, language engineers can create interpreters (language back-ends) described as meta-models with bi-directional model transformations.

At run time, the interpreter executes these transformations in sequence and produces transactions consisting of edit operations. Upon completion, it commits the transactions to the version history as cause-and-effect chains. Next, we introduce the main language concepts and features.

5.1 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) composed of objects. The language semantics steer the behavior of running programs. A program's run-time state, also a model, stores the results of program execution and user interactions.

For instance, Figure 8 shows the UML class diagram of LiveSML's meta-model. The static metamodel (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 registers how often it has resided in each state (the count).

```
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 9. Cascade definitions of LiveSML's meta-model

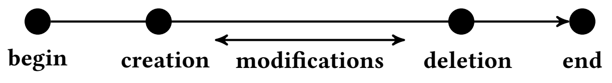


Figure 10. Object lifeline, events and modifications

The notation for meta-models resembles object-oriented programming, as shown in Figure 9. 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.

5.2 Actions, events and transformations

Cascade is designed to express run-time transformations with explicit effects. As a back-end language, it 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 can be created, modified or deleted. A *trigger* is an input event that has no direct effect, but can schedule other events, side-effects that happen afterwards. A *signal* is an output event that flags an occurrence such as an exception or an error.

5.2.1 Objects. Objects have a limited life span. Instead of operating on objects directly, events work on object *lifelines*, as shown in Figure 10. The life span of an object begins before its creation and ends after its deletion. Any number of changes may happen in between. These life stages are called *issued*, *bound* and *retired*.

5.2.2 Effects. The basic unit of change, called **`effect`**, offers a parameterized abstraction for scripting and reuse. Effects are bi-directional model transformations whose body is an edit script. Each effect has parameters, type-value pairs separated by commas that determine its scope. Figure 11 shows an example.

5.2.3 Creation. Creation effects are used to create new objects of a certain class. For instance, Figure 11 shows a partial specification of the Machine class. We use the REPL to create a new state machine called "doors" as shown in

```
 1 class Mach {
 2   String name;
 3   Set<State> states;
 4   Set<MachInst> instances;
 5
 6   effect Create(future Mach m,
 7       String name) {
 8     m = new Mach();
 9     m.name = name;
10     m.states = new Set<State>();
11     m.instances =
12       new Set<MachInst>();
13   }
```

```
14   inverse effect Delete(past Mach m,
15       String name = m.name) {
16     delete m.instances;
17     delete m.states;
18     m.name = null;
19     delete m;
20   } pre {
21     foreach(State s in m.states) {
22       State.Delete(s, s.name, m); }
23     foreach(MachInst mi in m.instances) {
24       MachInst.Delete(mi, m); }
25   }
26   ...
```

Figure 11. Partial Cascade specification of the Mach class

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

(a) Creating a machine

```
print m; ↩
machine doors
```

(b) Obtaining feedback

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

(c) Generated transaction

Figure 12. Creating state machine from the REPL

Figure 12a. For conciseness, we will omit declaring variables from now on. This command calls the Create effect (lines 6–13). Note the ↩ symbol indicates REPL input (pressing the return key), and its absence indicates output the interpreter gives in response. We verify the results by reading the output from the REPL in Figure 12b. The interpeter also generates the changes that have occurred. The transaction shown in Figure 12c is a short-hand for encapsulated edit operations.

5.2.4 Subject. The first parameter of an effect, called *subject*, is always a reference to the object that is subject to change. The subject can optionally be preceded by an additional keyword that provides guarantees about its life before and after execution. The **`future`** keyword, used only in creations, denotes the subject must be issued and will be bound afterwards. The **`past`** keyword, used only in deletions denotes a bound subject will be retired afterwards. The lack of a keyword signifies it will continue to exist.

5.2.5 Parameters. There are two kinds of additional parameters that may follow the subject in the signature. Constant parameters are inputs that enable passing values such as an **`int`**, **`bool`**, **`String`**, **`enum`** or object reference. Change parameters enable updating the value of an object field from an old to a new value. In Figure 11, both parameters of the Create effect (lines 6–7) are constant parameters. The Delete effect (line 15) also has a change parameter. It indicates transactions will store the old and the new value of name field.

5.3 Side-effects and causal tranformations

5.3.1 Side-effects. The **`pre`** and **`post`** clauses enable scheduling effects before and after an event. Side effects can be used to create modular constructors and destructors that keep the syntax and the run-time states consistent. The **`post`** clause enables creating additional objects, booting up systems and defining effects of user interaction. These clauses can contain if-statements and while loops that read values and schedule events, but cannot modify values directly. Triggers only have a post-clause. Signals have no side-effects.

5.3.2 Begin statement. For weaving side-effects together, the `begin` and `end` statements issue objects references and revoke their validity. For instance, "`begin State s;`" is a statement that issues a new reference to a State object. Afterwards, we can schedule its creation.

5.3.3 Deletion. By design, every object that can be created can also be deleted. Unlike creations, which work on "blanks slates", deletions must account for ownership and consistency. The `pre` clause enables performing clean up tasks such as deleting owned data, removing aliases, and shutting down entire systems. For instance, calling `Mach.Delete(m)` from the REPL does not only delete a machine, but also cleans up the every state and any running instance it owns, as defined by the pre-clause on lines 20–25 of Figure 11. We demonstrate how LiveSML handles deletion in Section 7.

5.3.4 Inverse. Inverse effects, indicated by the `inverse` keyword, perform conceptually opposite operations to their preceding effects. Effects and their inverses must have compatible signatures. Create and Delete in Figure 11 are inverse effects. Create's `future` subject and constant parameters match Delete's `past` subject and change parameters. At run time, an inversion entails creating an opposite effect that can roll back a transaction, undoing its effects. An `invertible` effect, typically a setter, is its own inverse. For such an effect, an inversion matches the constant subject, and swaps the old and new values of the change parameter.

5.4 Design considerations

5.4.1 Root cause analysis. The design decision, that all change must be explicit, adds some verbosity but ensures events can always be related to their Cascade specification. We observe that default code for effects and inverses may be generated, and explicit ownership enables static analysis.

5.4.2 Consistency. To ensure bi-directionality, deletions and removals must also be explicit. Therefore objects cannot be garbage-collected. 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. Fields must all have default values before deletion.

5.4.3 Liveness. Cascading changes can introduce liveness into DSLs. By adding relationships between coding actions and run-time effects, language engineers can improve input and feedback mechanisms that help programmers make gradual changes and observe differences in behavior. This has far reaching implications for the language designs of DSLs, as we will demonstrate in Sections 7, 8 and 9.

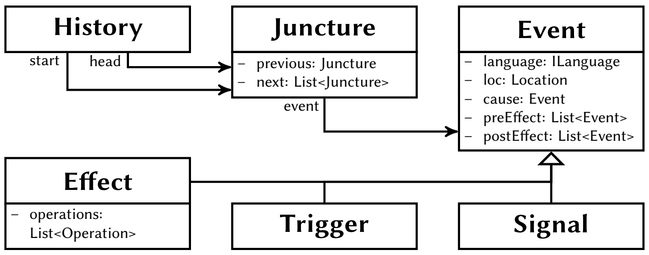


Figure 13. Delta's history consists of cause-and-effect chains

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

Figure 14. C# pseudo-code of Delta's event scheduler

6 Cascade Compiler and Runtime

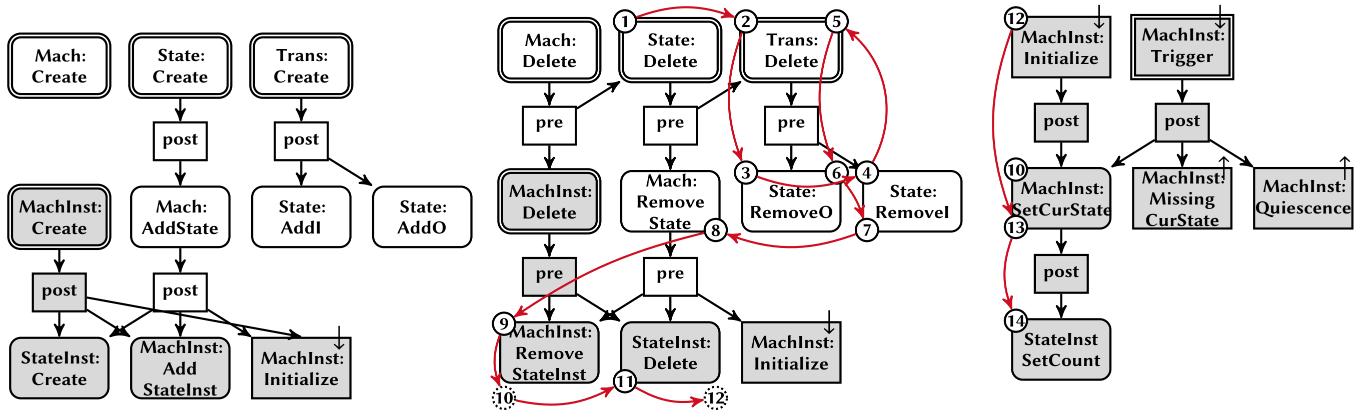
The Cascade framework consists of a compiler written in Rascal [8], and Delta, a runtime written in C#. The compiler translates Cascade specifications into language modules that integrate with Delta's extensible engine. Figure 6 gives an overview that illustrates how the main components process and transform events.

Each generated DSL interpreter (or language) consists of three sub-packages: 1) Model contains the classes of the meta-model; 2) Operation contains the classes representing events; and 3) Runtime contains components that process events and transform models. Key runtime components are the generator, and pre- and post-migrators, which generate edit operations and handle side-effects. Delta's engine has three main components. The dispatcher manages a set of languages, and determines which one handles an event. The patcher executes edit operations, maintains the heap and updates the version history. The scheduler determines the order in which events are scheduled, generated and migrated. Next, we explain how non-linear event scheduling works.

6.1 Scheduling events

The engine generates transactions in the form of cause-and-effect chains, as illustrated by Figure 13. Histories consists of junctures, branching points in time signifying events.

When called, the scheduler binds an event to a specific subject. We sketch the recursive algorithm that schedules each event in Figure 14. 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).



(a) Dependencies between creation events **(b)** Dependencies between deletion events **(c)** Dependency of run-time triggers

Figure 15. LiveSML: Static dependency graphs of events, and an example run-time state migration

The patcher immediately commits the transaction to the history, before the operations go stale (line 5). Afterwards, post-migration schedules any events that need to happen afterwards (line 6). When each of those events completes, the event itself completes.

6.2 Implementation

The compiler consists of 3285 LOC of Rascal. Delta consists of 11.5 KLOC of C#. Delta's main parts are the edit operations interpreter (1745 LOC), the runtime (7288 LOC), and the REPL language (2603 LOC). Cascade is available under the 2-clause BSD license: https://github.com/vrozen/Cascade.

7 Live State Machine Language

We investigate how to express the design of LiveSML [24]. LiveSML exemplifies run-time state migrations with one-to-many relationships. Its semantics introduce dependencies between definitions of machines and states and their instances. Changes to definitions potentially have many side-effects on the run-time state. Because this state is not known a priori, run-time state migrations have to account for many eventualities. Using Cascade, we create an interactive prototype that reproduces the behavior of the original Java implementation. We demonstrate its interpreter accounts for run-time eventualities by reproducing the scenario of Section 2.1.

7.1 Event-based language design

We create an event-based language design. In addition to its meta-model, shown in Figure 8, we design its run-time transformations. Figure 15 schematically depicts the static dependencies between creation events, deletion events and run-time triggers. Events, shown as rounded rectangles, work on the syntax (white) and the run-time state (gray). Interactive events (double line) are coding actions (white) and user interactions (gray) with side-effects (single line). Arrows indicate if side-effects happen before (pre) or after (post) an event. Converging arrows indicate reuse in distinct scenarios.

Creation. The programmer begins with an empty state machine by creating one. At any moment, they can add states to a program and transitions between states by creating new states and transitions. They can also run a machine at any point in time. Each running machine separately keeps track of its visit counts. Therefore, creating a new machine also instantiates each state. Running machines update their bookkeeping when adding a new state to their machine definition. Afterwards, each running machine reinitializes (↓), since it may not have a current state yet.

Deletion. Of course, programmers can also delete a machine. Each machine cleans up its states and running instances. Deleting a state has side-effects that also remove and delete every transition from its inputs and outputs. In addition, removing a state also removes and deletes state instances from every running machine. Finally, removing the current state of a running machine migrates its to the first state in its definition.

Run-time triggers. Triggering (↓) a running machine can cause a transition that sets a new current state. However, it can also result in the signals (↑) missing state or quiescence. These signals do not cause any change, but do provide feedback to the user. When setting a new current state, its count is also increased by one.

Prototype. The Cascade implementation of LiveSML counts 213 LOC. Compiling the sources results in 2204 LOC of generated C#. Next, we will apply the fully generated prototype.

7.2 Live programming scenario

We reproduce the run-time state migration of Section 2.1. Instead of parsing an SML program, we simulate sequential coding actions, user interactions, and feedback directly from the REPL. Figure 16 shows the REPL commands and feedback that simulate the scenario.

We first create a new machine *doors* that contains a *closed* state (Figure 16a). Using the **print** command, we call the pretty printer. We obtain feedback and verify the syntax of

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

(a) Creating the program

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

(b) Runing the program

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

(c) Completing the program

```
print mi; ↩
machine doors
  [open] [lock]
  closed : 1 *
  opened : 0
  locked : 0
```

(d) Updated run-time state

```
MachInst.Trigger(mi, "lock"); ↩
print mi; ↩
machine doors
  [unlock]
  closed : 1
  opened : 0
  locked : 1 *
```

(e) Locking the door

```
State.Delete(s3, "locked", m); ↩
print mi; ↩
machine doors
  [open]
  closed : 2 *
  opened : 0
```

(f) Deleting the locked state

Figure 16. LiveSML scenario simulated from the REPL

the DSL program is as expected. Although the program is not yet complete, we already run it. We create an instance, and use the **print** command to observe that, initially, its current state (*) is closed (Figure 16b).

We now complete the program (Figure 16c) by adding *opened* and *locked* states, and the transitions between them. Behind the scenes, several side-effect have occurred. We inspect the running program has also been updated (Figure 16d). The text between brackets denote "buttons" for the available actions. Users can now *open* or *lock* the closed door.

We simulate an action that locks the door (Figure 16e). Finally, we delete the locked state (Figure 16f). As expected, this causes a run-time state migration, setting the current state to closed, and increasing its count by one.

The resulting transaction, described in more detail in Appendix B, is super-imposed on the design on Figure 15. Its generated control flow traverses events that affect both the syntax and the run-time state. Note that, the edit operations of the deletion itself, actually happen last.

7.3 Analysis

Compared to the original LiveSML, which counts 1217 LOC of hand-written Java, our prototype is significantly smaller (213 LOC)[2]. Cascade addresses the main shortcoming of the Run-time Model Patching (RMPatch) approach that expresses run-time state migrations as hard-coding visitors on edit operations, which is time consuming and error-prone [24]. Instead, Cascade expresses them on the meta-level. As a result, LiveSML's modular design is more concise and maintainable.

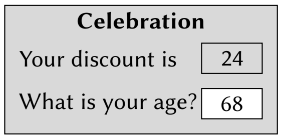
8 Live Questionnaire Language

The Questionnaire Language (QL) is a DSL for expressing interactive digital questionnaires. Originally designed for the

[2]https://github.com/vrozen/Cascade/tree/main/LiveSML

```
form Celebration {
  "Your discount is" :
    int discount = age/2-10
  "What is your age?" :
    int age
}
```

(a) Celebration form


Celebration
Your discount is 24
What is your age? 68

(b) Example filled-out form

Figure 17. Forms that calculates an age dependent discount

Dutch tax office, this DSL has since served as a benchmark for generic language technology, e.g., the Language Workbench Challenge [7]. We study LiveQL, a language variant that enables simultaneously designing and answering forms [20]. We focus on the liveness properties and trickle effects at the heart of its semantics. In particular, we aim to reproduce the behavior that propagates the effects of giving answers. Using Cascade, we express LiveQL and create a language prototype. We use its REPL to simulate a run-time scenario of an example that demonstrates a fixpoint computation.

8.1 Questionnaire Language

Forms consists of sequences (or blocks) of two kinds of statements: questions and if statements. Figures 17 shows a from that expresses an age-based discount, and an answered form.

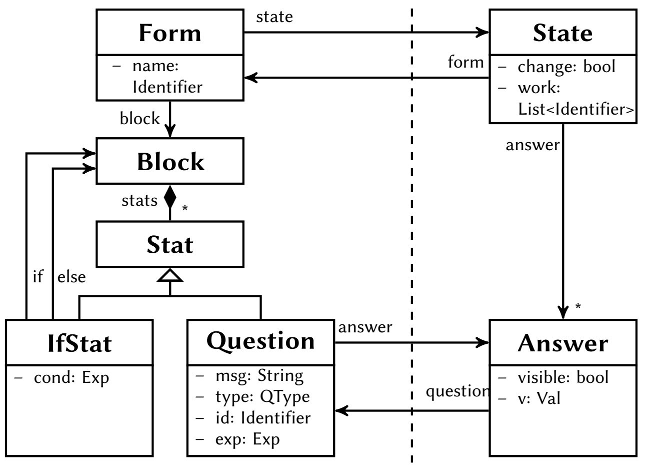
8.1.1 Questions. Each question consists of a textual message the user sees, a question type (int, str or bool), and a variable name that can be used to reference the question's answer. By default, questions are answerable. The question "what is your age?" is answerable. Users answer questions by supplying a value of the specified question type. In this case, age requires an int value, for instance 68.

However, when assigned with an expression, questions become *computed*. Instead of prompting the user to answer the question, the form computes the answer by evaluating the expression. In the example, discount is a computed answer. Its computed value is 24.

8.1.2 If statements. Conditional questionnaire sections can be designed using if statements. Each if consists of a condition (a boolean expression), an if-block and an optional else-block. The user sees the statements nested in the if-block if the condition is true, and those in the else-block otherwise. Statements that have an expression referencing a variable have *data dependency* on that variable. Statements nested inside an if-block have a *control dependency* on each variable referenced in the condition. Here, we omit an example.

8.1.3 LiveQL. Originally, QL required that users answer questions in a top-down manner [7]. Each statement could only refer back to variables whose value have been previously given or computed.

LiveQL relaxes this requirement by enabling forward references, and allowing changes to running forms [20]. These changes introduce two forms of liveness. First, when the programmer adds, removes or changes statements, this affects the running form. Second, when a user answers a question,



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

Figure 18. Static and run-time meta-models of LiveQL

```
package LiveQL {
  class State {
    ref Form f;
    List<Answer> ans;
    bool change;
    List<Identifier> work;

    trigger TriggerID(State s, Identifier i) {
      SetChanged(s, false); //First, reset the changed flag.
      PushWork(s, i);       //Next, add the identifier to the work queue.
      DoWork(s);            //Schedule the work, which may cause re-evaluations.
      WhileChange(s, i); }   //Finally, check if there is more change.

    trigger WhileChange(State s, Identifier i) {
      if(s.change) {     //If a change has happened as a result of re-evaluation
        TriggerID(s, i); //continue the computation
      } else {           //otherwise
        Done(s, i);      //signal done.
      } }

    signal Done(State s, Identifier i);
```

Figure 19. LiveQL contains a fixpoint computation

the form re-evaluates dependent computed questions and if statements. As a result, answers may update and sections of the form can become visible or invisible.

8.2 Language design

We investigate how Cascade helps to express the behavior of LiveQL, in particular the trickle effects that result from answering questions. The meta-model of LiveQL, shown in Figure 18, is based on the original Java implementation [20]. The run-time meta-model extends the static meta-model with information about the current state of the form, such as answers to questions and visibility.

The key to expressing trickle effects is defining a fixpoint computation that schedules future events in sequence from the body of a trigger. When answering a question, dependent computed answers and if-statements recompute until no more changes can be observed. Figure 19 illustrates the main events. When the value of an identifier updates, `TriggerID` is called. After performing work, which potentially causes changes, the check of `WhileChange` determines if the computation completes or continues to propagate changes.

```
1 Form.Create(f, "Celebration"); ↩
2 Question.Create(q1, f, "Your discount is", QType.Int, "discount");↩
3 Question.SetExpression(q1, f, "age/2-10"); ↩
4 Question.Create(q2, f, "What is your age?", QType.Int, "age"); ↩
```

(a) Creating the Celebration form

```
Question.GiveAnswer(q2, f, "68"); ↩
```

(b) Answering the age question

```
print f; ↩
form Celebration {
  "Your discount is" :
    int discount = age/2-10
      ==> undefined
  "What is your age?" :
    int age ==> undefined
}
```

(c) Unanswered form

```
print f; ↩
form Celebration {
  "Your discount is" :
    int discount = age/2-10
      ==> 24
  "What is your age?" :
    int age ==> 68
}
```

(d) Answered form

Figure 20. LiveQL scenario simulated from the REPL

8.3 Prototype live programming environment

We create a textual DSL prototype, an interpreter with a built-in REPL. Its Cascade specification counts 1044 LOC[3]. Compiling the sources results in 8189 LOC of generated C#. Most components of the prototype are fully generated. We add the following components, which amounts to a total of 916 LOC hand-written C#.

The pretty-printer enables inspecting the syntax and run-time state from the REPL. We add an expression evaluator and two small helper classes for: 1) performing lookups for use-def relationships of variables; 2) collecting conditions of questions; and 3) evaluating the expressions of questions and if-statements. We use ANTLR 4 to create a QL parser that generates ASTs of programs and expressions. To bring these ASTs under management of Delta, we create a Builder that generates Cascade events for recreating the ASTs.

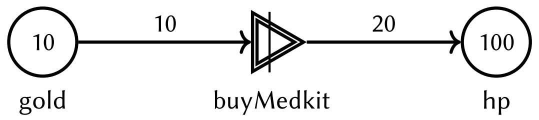
8.4 Live programming scenario

We demonstrate a trickle effect in a live programming scenario that reproduces the Celebration example of Figure 17. After creating the form, answering the question `age` with `68` should result in the `discount` becoming `24`.

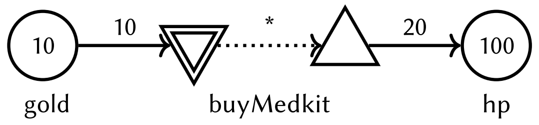
Using our prototype, we simulate coding actions and user interaction from the REPL, as shown in Figure 20. We begin with the coding actions shown in Figure 20a. First, we create a new form `f` called Celebration (line 1). We add a new question `q1` to form f with message "Your discount is", introducing the variable `discount` of type int (line 2). To make q1 a computed question, we set its expression to `age/2-10` (line 3). Finally we add the second question `q2` introducing int `age` (line 4).

LiveQL programs automatically run one instance. We verify the program runs, and observe `discount` and `age` are initially undefined, as shown in Figure 20c. Next, we answer question `q2` and give answer `68`, as shown in Figure 20b. Finally, we verify the value of `discount` has indeed become `24`. Figure 20d indicates the change has been correctly propagated. For conciseness, we omit the generated transaction.

[3] https://github.com/vrozen/Cascade/tree/main/LiveQL



(a) Mechanism for exchanging gold for health (what designers see)



(b) Desugared version of the diagram (what the engine runs)

Figure 21. Diagram showing an excerpt of the internal economy of Johnny Jetstream (adapted from van Rozen [21])

8.5 Analysis

Our results demonstrate Cascade helps express the trickle effects of LiveQL as a concise fixpoint computation. Creating the prototype, including its helper classes, cost approximately one working day, with only the experience of LiveSML. The original Java implementation, which measures 6179 LOC, also includes a visual front-end. The new prototype is significantly smaller, measuring 816 LOC.

9 Live Machinations

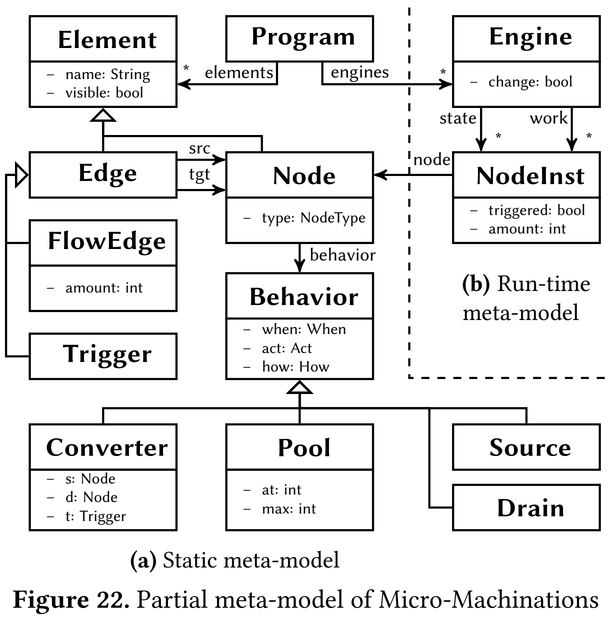
Machinations is a visual notation for game design that foregrounds elemental feedback loops associated with emergent gameplay [1, 1]. Micro-Machinations (MM) is a textual and visual programming language that addresses several technical shortcomings of its evolutionary predecessor. In particular, MM introduces a live programming approach for rapidly prototyping and fine-tuning a game's mechanics [23], and accelerating the game development process.

We study the design of the MM library (MM-Lib), including its run-time bahavior and state migrations. In particular, we explore how Cascade helps to express live desugaring. We create a visual prototype using Cascade and the Godot game engine. Vie is a tiny live game engine for simultaneously prototyping and playtesting a game's mechanisms. In live programming scenario of a simple game economy, we demonstrate Vie correctly desugars converters.

9.1 Micro-Machinations

Micro-Machinations programs, or diagrams, are directed graphs that can control the internal economy of running digital games. When set in motion through runtime and player interactions, the nodes act by pushing or pulling economic resources along its edges.

Figure 21a shows a mechanism in the internal economy of Johnny Jetstream, a 2D fly-by shooter [21]. Two *pool* nodes, shown as circles, abstract from the in-game resources, gold and health (hp). The integers inside represent current amounts. The edges are *resource connections* that define the rate at which resources can flow between source and target nodes. BuyMedkit is an interactive *converter* node, appearing



(b) Run-time meta-model

(a) Static meta-model

Figure 22. Partial meta-model of Micro-Machinations

as a triangle pointing right with a vertical line through the middle. This converter consumes 10 gold and produces 20 hp.

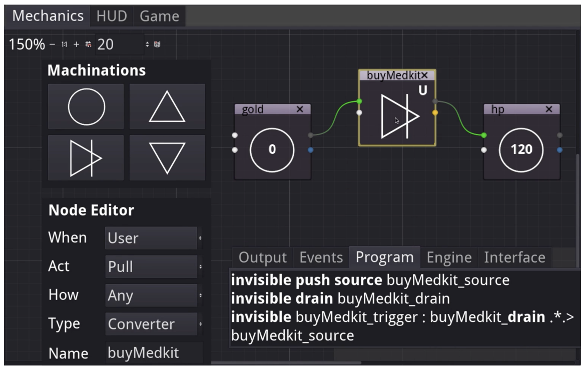
Converters are so-called syntactic sugar, a convenience notation, which translates into a simpler elements for efficient processing. Figure 21b shows that converters can be rewritten as a combination of a drain, a trigger and a source. When the drain node consumes the costs of the conversion, the trigger activates the source node, which then produces the benefits. During the translation, the inputs of the converter connect to the drain, and the outputs to the source.

Designers can simultaneously prototype and playtest running economies, e.g., by activating mechanisms or modifying node types. Therefore, desugarings must also happen live.

9.2 Language design

We investigate how Cascade helps to express the liveness of MM's core language features, focusing on live desugaring of converters in particular. As a starting point, we analyze the C++ implementation of MM-Lib, an embeddable script engine for MM [23]. Figure 22 shows a partial meta-model based on MM-Lib. As before, we express the dependencies between the abstract syntax and the run-time state. An engine, which instantiates a program, tracks the current amounts of pool nodes and which nodes are triggered for activation. Through a combination of effects and helper methods, it evaluates how the resources flow when nodes activate.

The solution for desugaring converter nodes introduces invisible elements that implement its behavior. When a node becomes a converter, a series of transformations immediately generate 1) a source, a trigger and a drain; 2) incoming edges to the drain; and 3) outgoing edges to the source. In addition, running engines obtain new node instances used for evaluating flow rates. Changes to a converter node are delegated its source and drain nodes. Changing the node's



When User
Act Pull
How Any
Type Converter
Name buyMedkit

```
invisible push source buyMedkit_source
invisible drain buyMedkit_drain
invisible buyMedkit_trigger : buyMedkit_drain .*.>
buyMedkit_source
```

Figure 23. Prototyping and playtesting a mechanism in Vie

behavior to another type, cleans up these invisible elements, and also removes node instances from running engines.

9.3 Vie: a tiny live game engine

We use Cascade and the Godot game engine to create a prototype that implements the design of MM. Vie is a tiny live game engine for simultaneously prototyping, playtesting and fine-tuning a game's design. MM's implementation in Cascade counts 1542 LOC[4]. Compiling its sources generates an interpreter, 11.7 KLOC of generated C#. We augment Engine with two helper classes, EvalContext (153 LOC) and Flow (29 LOC), for storing temporary run-time data.

Instead of using its built-in REPL, we create a visual front-end using Godot (v3.5.1). We leverage the GraphNode and GraphEdit framework, and program C# classes for connecting Cascade events to UI events. In another paper we further detail how we manually create this front-end [22].

9.4 Live programming scenario

We reproduce the behavior of the mechanism shown in Figure 11. Using Vie, we perform a sequence of prototyping and playtesting actions that demonstrate the behavior of MM, including live desugaring and run-time state migrations.

We first recreate the diagram using Vie's visual editor, shown in Figure 23. Next, we trigger the converter buyMedit by clicking on its center. The UI shows visual feedback. We observe the engine succeeds (yellow) in activating its drain, trigger and source. The nodes consume and produce resources at the expected rates (green). The textual view on the program, shows the invisible elements. Finally, we change the node type of the converter to pool. The resulting transaction is a long cause-effect-chain that cleans up the desugared converter and migrates the run-time state.

9.5 Analysis

MM-Lib measures 21.2 KLOC of C++. In comparison, Vie does not yet support every feature, e.g. modules. However, at a mere 1542 LOC, its Cascade specification is considerably more concise. Due to its representation of effects, Vie solves two limitations of MM-Lib. First, it adds traceability of cause-and-effect for all actions. Second, it expresses the effects of resource propagation and triggers as a fixpoint computation. Vie is more extensible and maintainable. Combining Godot with Cascade is straightforward. Since both have event APIs, and Godot support C#, they integrate well. Our effort went mainly into creating the UI. A benefit of Godot is that Vie is a portable app (Windows, Linux, MacOS, iOS, Android).

[4]https://github.com/vrozen/Cascade/tree/main/LiveMM

10 Discussion

Cascade has compelling benefits for creating live programming environments. Using its notations and abstractions, language engineers can concisely express DSL run-time behaviors, and account for many migration scenarios, on the meta-level. They can ensure transformations and side-effects are correct by design. However, no automated contextual analysis is provided yet. At no additional cost, Delta generates a history that traces how and why every event happens, while ensuring the run-time state is correctly updated. Cause-and-effect chains are instrumental for exploratory live programming, omniscient debugging and version control. Because the generated interpreters are event-driven, they combine well with visual UIs, e.g., bowsers or game engines.

Of course there are also costs. For language engineers, bidirectional thinking is not straightforward. Language designs do not normally include run-time state migrations. Learning how bi-directional designs work takes time and practice.

We have validated Cascade against a limited number of existing DSLs. Further validation will require introducing new liveness to DSLs. Additionally, the compiler is a complex meta-program that bridges a wide conceptual gap. As a result, it undoubtedly still contains bugs we have not yet identified. To address this, we plan to create a test harness that automates testing features and prototypes.

Of course, the proposed combination of transformations, migrators and feedback mechanisms generalizes beyond Cascade. These abstractions can also be programmed using General Purpose Languages (GPLs). Our compiler targets C#. Unlike Cascade, GPLs do not support bi-directional transformations or generate cause-and-effect chains out of the box. Adding support requires a considerable engineering effort.

We have not studied using Cascade to add liveness to GPLs. The underlying execution models, e.g., program counters and stack frames, do not directly support liveness. GPLs require additional mechanisms such as probes to introduce liveness.

Cascade's generic REPL has limitations. Its mechanisms are low-level, and not suitable for DSL users. Furthermore, Cascade still lacks a debugger for exploring histories, inspecting cause-and-effect chains and tracing source locations. Debugging a DSL involves stepping through the generated C# code. Debugging transactions involves inspecting the notation on the REPL (e.g., Figure 24 in Appendix B).

Cascade's integration in C# is helpful for extending it functionality with helper methods. The lack of a formal semantics complicates analyses. We see opportunities for checking cyclic dependencies and the correctness of inverse effects.

Live programming with run-time state migrations is inherently inconsistent. An open challenge is identifying formal properties of liveness. Cascade introduces a local dependencies between events that have a global consistency of effects.

11 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 [16]. Tanimoto describes *levels of liveness* that help distinguish between forms of feedback in live programming environments [18]. 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 [11]. Ko describes *whyline*, a debugging mechanism for asking why-questions about Java program behavior [9].

Another approach is creating interpreters with a so-called Read-Eval-Print-Loop (REPL), a textual interface for executing commands sequentially [3] A REPL, by definition, lends itself naturally to exploration, incremental change and immediate feedback, each key ingredients to live programming. Interpreters created with Cascade have a built-in REPL and REPL-like APIs for designing DSLs with event-based input and feedback mechanisms, including visual ones.

Omniscient debugging, also called back-in-time debugging, is a form of debugging that allows exploring what-if scenarios by stepping forward and backward through the code [14, 15]. Such debuggers have been created for general purpose languages Java [15]. Retrofitting an omniscient debugger to an existing language can come at a considerable cost, redesign and implementation effort. Bousse et al. propose a meta-modeling approach for a generic debugger of executable DSLs that supports common debugging services for tracing the execution [4]. Cascade is also designed with omniscient debugging in mind. Cause-and-effect chains are a key data structure for creating omniscient debuggers.

The area of *modelware* is a technological space that revolves around the design, maintenance and reuse of models (or programs). 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, which expresses model deltas [2]. Van der Storm proposes creating live programming environments driven by "semantic deltas", based on this notation [20]. Van Rozen and van der Storm combine origin tracking and text differencing for textual model differencing [21].

Bi-directional Model Transformation (BX) is a well researched topic that intersects with several areas [6]. BX has impacted relational databases, model-driven software development [17], UIs, 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 [5].

The study of live modeling with run-time state migrations has initially focused on fine-grained patching with edit operations [24]. Constraint-based solutions instead focus on correct states with respect to a set of constraints [19], a course-grained approach that omits fine-tuning. Sanitization solutions regard run-time state migrations as a way to fix what is broken [25]. We instead take the position that they are an integral part of the language semantics.

Until now, no solution could explain why a particular migration happened. Cascade is a BX solution that addresses both *how* and *why*, for precise feedback about root causes and finger sensitive fine-tuning.

Cascade is the first language-parametric and generic approach for creating DSLs that leverages a bi-directional transformations for live programming. Compared to existing approaches, it adds a scheduling mechanism for defining complex deterministic side-effects. To the best of our knowledge, no other system exists that can generate run-time state migrations from meta-descriptions as cause-and-effect chains.

12 Conclusions and future work

We have addressed the lack of enabling technology for creating live programming environments. We have proposed Cascade, a meta-language for expressing DSLs with interface- and feedback-mechanisms that drive live programming. In three case studies, we have explored expressing the liveness features of LiveSML, LiveQL and Machinations. We have demonstrated how to express gradual change, run-time state migrations, ripple effects and live desugarings. Our results show that an explicit representation of change is instrumental for how these languages are built, and that cause-and-effect relationships are vital for delivering feedback.

12.1 Future work

In future work, we will investigate how to create a reusable omniscient debugger for change-based DSL environments.

Schema or program modifications require both instance migrations and view adaptations. We will further investigate how to express coupled transformations [10], e.g., for Vie.

Live programming requires interactive visual interfaces, and generic language technology to create them [12]. We will investigate how to automate the development of UIs that leverage game engine technology [22].

Acknowledgments

I thank the anonymous reviewers for their insightful comments, which helped improve this paper. Additionally, I would like to express my appreciation to the Agile Language Engineering (ALE) associate team (Inria & CWI) for their valuable feedback and support.

References

[1] Ernest Adams and Joris Dormans. 2012. *Game Mechanics: Advanced Game Design.* New Riders.

[2] Marcus Alanen and Ivan Porres. 2003. Difference and Union of Models. In *«UML» 2003 (LNCS, Vol. 2863).* Springer.

[3] L. Thomas van Binsbergen, Mauricio Verano Merino, Pierre Jeanjean, Tijs van der Storm, Benoît Combemale, and Olivier Barais. 2020. A Principled Approach to REPL Interpreters. In *Onward! 2020.* ACM.

[4] Erwan Bousse, Dorian Leroy, Benoît Combemale, Manuel Wimmer, and Benoit Baudry. 2018. Omniscient Debugging for Executable DSLs. *J. Syst. Softw.* 137 (2018).

[5] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. 2010. JTL: A Bidirectional and Change Propagating Transformation Language. In *Software Language Engineering, SLE2010 (LNCS, Vol. 6563).* Springer.

[6] 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 (LNCS, Vol. 5563).* Springer.

[7] Sebastian Erdweg, Tijs van der Storm, et al. 2013. The State of the Art in Language Workbenches. In *Software Language Engineering, SLE 2013 (LNCS, Vol. 8225).* Springer.

[8] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Code Analysis and Manipulation, SCAM 2009.* IEEE.

[9] Amy J. Ko and Brad A. Myers. 2004. Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior. In *Human Factors in Computing Systems, CHI 2004.* ACM.

[10] Ralf Lämmel. 2016. Coupled Software Transformations Revisited. In *Software Language Engineering, SLE 2016.* ACM.

[11] Sean McDirmid. 2013. Usable Live Programming. In *New Ideas in Programming and Reflections on Software, Onward! 2013.* ACM.

[12] Mauricio Verano Merino, Jurgen J. Vinju, and Tijs van der Storm. 2020. Bacatá: Notebooks for DSLs, Almost for Free. *Art Sci. Eng. Program.* 4, 3 (2020), 11.

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

[14] Guillaume Pothier and Éric Tanter. 2009. Back to the Future: Omniscient Debugging. *IEEE Softw.* 26, 6 (2009).

[15] Guillaume Pothier, Éric Tanter, and José M. Piquer. 2007. Scalable Omniscient Debugging. In *Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007.* ACM.

[16] 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.

[17] Perdita Stevens. 2010. Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. *Softw. Syst. Model.* 9, 1 (2010).

[18] Steven L. Tanimoto. 2013. A Perspective on the Evolution of Live Programming. In *Workshop on Live Programming, LIVE 2013.* IEEE.

[19] Ulyana Tikhonova, Jouke Stoel, Tijs van der Storm, and Thomas Degueule. 2018. Constraint-based Run-time State Migration for Live Modeling. In *Software Language Engineering, SLE 2018.* ACM.

[20] Tijs van der Storm. 2013. Semantic Deltas for Live DSL Environments. In *Workshop on Live Programming, LIVE 2013.* IEEE.

[21] Riemer van Rozen. 2015. A Pattern-Based Game Mechanics Design Assistant. In *Proceedings of the 10th International Conference on the Foundations of Digital Games, FDG 2015, 2015.* SASDG.

[22] Riemer van Rozen. 2023. Game Engine Wizardry for Programming Mischief. In *Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments, PAINT 2022.* ACM.

[23] Riemer van Rozen and Joris Dormans. 2014. Adapting Game Mechanics with Micro-Machinations. In *Proceedings of the 9th International Conference on the Foundations of Digital Games, FDG 2014.* SASDG.

[24] 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).

[25] Yentl Van Tendeloo, Simon Van Mierlo, and Hans Vangheluwe. 2019. A Multi-Paradigm Modelling Approach to Live Modelling. *Softw. Syst. Model.* 18, 5 (2019).

Table 1. Edit operations supported by Delta

Operation	**Inverse operation**
o_new (ID id, StringVal class)	**o_delete** (ID id, StringVal class)
o_set (Path p, Field f, Val nv,Val ov)	**o_set** (Path p, Field f, Val ov,Val nv)
l_insert (Path p, IntVal i, Val v)	**l_remove** (Path p, IntVal i, Val v)
l_push (Path p, Val v)	**l_pop** (Path p, Val v)
l_set (Path p, IntVal i, Val nv,Val ov)	**l_set**(Path p, IntVal i, Val ov,Val nv)
s_add (Path p, Val v)	**s_remove** (Path p, Val v)
m_add (Path p, Val k)	**m_remove** (Path p, Val k)
m_set (Path p, Val k, Val nv, Val ov)	**m_set** (Path p, Val k, Val ov, Val nv)

A Edit Operations

This appendix describes on a complete set of edit operations for manipulating commonly used data structures: objects, lists, sets and maps. Cascade's runtime Delta uses these operations to execute bi-directional model transformations.

A.1 Objects, heap and qualified names

Edit operations work on objects. A global store, or heap, stores the current program state as a collection of objects with fields and references. Values can have a base type such as **`int`**, **`String`**, **`bool`**.

Enums and classes introduce custom objects and values. The built-in datatypes `Set`, `List` and `Map` help to create object hierarchies, such as trees and graphs. Edit operations can perform lookups using qualified names or paths. The notation uses dots and brackets for lookups in the global store, maps and lists. For instance, `[|uuid://1|]` performs a lookup of an object with Unique Universal Identifier (UUID) 1, and `[|uuid://1|].cur` retrieves its field `cur`.

A.2 Supported edit operations

Delta supports the edit operations shown in Table 1. We briefly describe these operations, which work on objects, lists sets and maps. Figure 7a shows an example sequence of operations that partially recreate the example SML program shown in Figure 3.

```
root ① State.Delete([|uuid://26|], "locked", [|uuid://13|]) {
  pre ② Trans.Delete([|uuid://33|], [|uuid://26|], "unlock", [|uuid://16|]) {
    pre ③ State.RemoveOut([|uuid://26|], [|uuid://33]) {
      [|uuid://28|].remove([|uuid://33|]);
    }
    pre ④ State.RemoveIn([|uuid://16|], [|uuid://33|]) {
      [|uuid://17|].remove([|uuid://33|])
    }
    [|uuid://33|].src = [|uuid://uuid26|] ⟶ null;
    [|uuid://33|].evt = "unlock" ⟶ null;
    [|uuid://33|].tgt = [|uuid://uuid16|] ⟶ null;
    delete Trans [|uuid://33|];
  }
  pre ⑤ Trans.Delete([|uuid://32|], [|uuid://16|], "lock", [|uuid://26|]) {
    pre ⑥ State.RemoveOut([|uuid://16], [uuid://32]) {
      [|uuid://18|].remove([|uuid://32|]);
    }
    pre ⑦ State.RemoveIn([|uuid://26|], [uuid://32]) {
      [|uuid://27|].remove([|uuid://32|])
    }
    [|uuid://32|].src = [|uuid://uuid16|] ⟶ null;
    [|uuid://32|].evt = "lock" ⟶ null;
    [|uuid://32|].tgt = [|uuid://uuid26|] ⟶ null;
    delete Trans [|uuid://32|];
  }
  pre ⑧ Mach.RemoveState([|uuid://13|], [|uuid://26|]) {
    pre ⑨ MachInst.RemoveStateInst([|uuid://19|], [|uuid://29|], [|uuid://26|]){
      pre ⑩ MachInst.SetCurState([|uuid://19|], null, [|uuid://29|]) {
        [|uuid://19|].cur = [|uuid://uuid29|] ⟶ null;
      }
      [|uuid://20|][[|uuid://26|]] = [|uuid://29|] ⟶ null;
      [|uuid://20|].remove([|uuid://26|]);
    }
    pre ⑪ StateInst.Delete([|uuid://29|], [|uuid://26|]) {
      [|uuid://29|].def = [|uuid://26|] ⟶ null;
      [|uuid://29|].count = 1 ⟶ 0;
      delete StateInst [|uuid://29|];
    }
    pre ⑫ MachInst.Initialize([|uuid://19|]) {
      post ⑬ MachInst.SetCurState([|uuid://19|], [|uuid://21|]) {
        [|uuid://19|].cur = [|uuid://21|];
        post ⑭ StateInst.SetCount([|uuid://21|], 2, 1) {
          [|uuid://21|].count = 1 ⟶ 2;
        }
      }
    }
    [|uuid://14|].remove([|uuid://26|]);
  }
  [|uuid://26|].name = "locked" ⟶ null;
  delete Set<Trans> [|uuid://27|];
  delete Set<Trans> [|uuid://28|];
  delete State [|uuid://26|];
}
```

Figure 24. Cause-effect chain that deletes the locked state

A.2.1 Objects. The **o_new** operation creates a new object with identifier id of a particular class. The new object will have all of its fields set to default values. Its inverse operation **o_delete** has the exact opposite effect, and deletes an object with identifier id. Deletions include the class parameter and they require that each fields of the object has default values. Without this, the operation cannot be reversed. Finally, to set the values of fields, the **o_set** operation, replaces the value ov of field f of the object denoted by path p by a new value nv. This operation is its own inverse, swapping the old and the new values. Figure 7a shows examples.

A.2.2 List. Specialized operations that only work on list objects of type List<X> are the following. The operations **l_insert** and **l_remove** are each other's inverse. These operations respectively insert or remove a value v at an index i in the list denoted by path p.

To modify a value in a list, the invertible operation **l_set**, replaces an old value ov by a new value nv at index i in the list denoted by path p. For convenience, **l_push** inserts a value v at the tail of an existing list, and **l_pop** removes it, without specifying the index.

A.2.3 Set. Two operations work on set objects of type Set<X>. The inverse operations **s_add** and **s_remove** respectively add or remove a value v of type X in an existing set denoted by path p.

A.2.4 Map. Operations that work on map objects of type Map<K,V> are the following. The operations **m_add** and **m_remove** respectively add or remove a map record denoted by key k of type K in an existing map denoted by path p. To ensure correctness, the initial and final value in a record must be defaults. To update a map record, the operation **m_set** replaces an old value ov by new value nv in the record denoted by key k in the map denoted by path p.

B LiveSML: Cause-and-effect chain

We detail the results of Section 7, which reproduces the example LiveSML live programming scenario of Section 2.1. Figure 24 shows the cause-and-effect chain that results from deleting the locked state. The numbers appearing in the circles coincide with those in the generated control flow that is superimposed on the static dependency graph of Figure 15.

Received 2023-07-07; accepted 2023-09-01