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 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.”
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)

\(^1\)https://github.com/vrozen/Cascade

---

### Table 1: The REPL language enables using live languages

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
<th>command</th>
</tr>
</thead>
<tbody>
<tr>
<td>do</td>
<td>perform a coding action or user action</td>
<td>call event</td>
</tr>
<tr>
<td>undo</td>
<td>roll back a transaction</td>
<td>undo</td>
</tr>
<tr>
<td>perceive</td>
<td>inspect state</td>
<td>print</td>
</tr>
<tr>
<td>play</td>
<td>play a history cause-effect-chains</td>
<td>future feature</td>
</tr>
<tr>
<td>rewind</td>
<td>unroll a history cause-effect-chains</td>
<td>future feature</td>
</tr>
<tr>
<td>view effect</td>
<td>display a cause-effect-chain textually</td>
<td>(output)</td>
</tr>
<tr>
<td>find cause</td>
<td>inspect a textual cause-and-effect-chain</td>
<td>(output)</td>
</tr>
</tbody>
</table>

---

![Figure 2: Static and run-time metamodels of LiveSML](image)

![Figure 3: Cascade definitions of LiveSML’s metamodels](image)

whose semantics steer the program’s behavior. The run-time state is also a model, storing the results of user actions.

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.
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.

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.
void schedule(Patcher p, Dispatcher d, Event e) {
    foreach (Event pre in d.preMigrate(e)) { schedule(pre); }
    d.resolve(e);
    foreach (Event post in d.postMigrate(e)) { schedule(post); }
    p.commit(e);
}

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, are scheduled first (line 3). Only when the recursive preside-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: open, 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 removes 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.

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.
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.

```plaintext
LiveSML.State.Create(s2, "opened", s3); ←
LiveSML.State.Create(s3, "locked", s4); ←
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.

```plaintext
print s; ↔
```

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.

```plaintext
print s;
```

3.2.4 Locking the door. The user locks the door as follows.

```plaintext
LiveSML.MachInst.Trigger(mi, "lock"); ↔
print s; ↔
```

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.

```plaintext
LiveSML.State.Delete(s3, "locked", s1); ↔
print s; ↔
```

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.

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 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


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.name = name;
      m.instances = new Set<MachInst>();
      m.states = new Set<State>();
    }
    inverse effect Delete(past Mach m, String name = m.name) {
      m.name = null;
      delete m.instances;
      delete m;
    }
    foreach {
      foreach (State s in m.states) { State.Delete(s, s.name, m); }
      foreach (MachInst mi in m.instances) { MachInst.Delete(mi, m); }
    }
    inverse 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);
    }
  }
}

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.

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.
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.

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.

```java
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);
      }
    }
    for (State s in mi.def.states) {
      begin StateInst si;
      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;
    }
    effect AddStateInst(MachInst mi, StateInst si, State s) {
      mi.sis[s] = si;
      post {
        Mach.AddMachInst(mi.def, mi);
      }
    }
    effect RemoveStateInst(MachInst mi, StateInst si, State s) {
      mi.sis.remove(s);
      post {
        Mach.RemoveMachInst(mi.def, mi);
      }
    }
    effect SetCurState(MachInst mi, StateInst cur = mi.cur) {
      mi.cur.def.o) {
        SetCurState(mi, nextState);
        StateInst nextState = mi.sis[mi.tgt];
      }
    }
    effect Initialize(MachInst mi) {
      if(mi.s.isValues() == si) {
        StateInst cur = mi.cur;
      }
      post {
        Mach.AddMachInst(mi.def, mi);
      }
    }
    effect Trigger(MachInst mi, String evt) {
      if(mi.cur == null) ( MissingCurrentState(MachInst mi); return; )
      foreach(Trans t in mi.cur.def() {
        if(t.evt == evt) {
          StateInst nextState = mi.sis[t.tgt];
          SetCurState(mi, nextState);
        }
      }
      Quiescence(mi);
    }
    signal MissingCurrentState(MachInst mi);
    signal Quiescence(MachInst mi);
  }
}
```