Model of a DSL for finite state machines

Ralf Lämmel

Software Languages Team
University of Koblenz-Landau, Germany

Abstract

We model a domain-specific language FSML for Finite State Machines (FSMs). The language model encompasses concrete a textual syntax, a term-based abstract syntax, a graph-based visual syntax, extra constraints for well-formedness, a simulation semantics, and a code-generation semantics for representing and executing FSMs in Java. The key motivation for the present model of FSML to be explainable exhaustively in terms of simple formalisms and idioms, all layered on top of Prolog. The FSML development is part of the SLEPRO project; see `https://github.com/slebok/slepro`.

Acknowledgement: This document and the underlying development are parts of a broader effort on language modeling and software language engineering. Collaboration with and feedback by Anya Helene Bagge (University of Bergen) and Andrei Varanovich (University of Koblenz-Landau) are gratefully acknowledged.

Version

0.1 as of 11 November 2014.

Website of this document:

`http://softlang.uni-koblenz.de/fsml/`

The page also links to supplementary repository locations. In particular, the model (code) of this document as well as alternative implementations of FSML are linked. The FSML model is maintained as part of the SLEPRO project: `https://github.com/slebok/slepro`

Contents

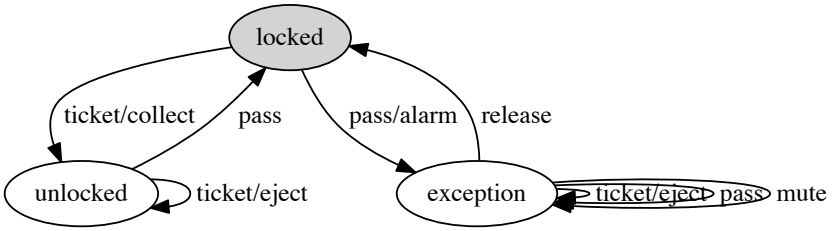

Figure 1: A finite state machine for turnstiles.

1. Introducing FSML

Consider the visual representation of a finite state machine (FSM) for a turnstile (as used in subway systems). The idea is, of course, that the customer is supposed to insert a ticket before passing the turnstile. Thus, the FSM assumes the initial state 'locked'. (The initial state is highlighted by using a filled ellipse.) The input 'ticket' (meaning the insertion of a ticket) causes a transition to the state 'unlocked' and the action 'collect'. In this new state, the input 'pass' (meaning the attempt to pass the turnstile) causes a transition back to the state 'locked'. If the input 'pass' was made in the state 'locked', then this causes a transition to the state 'exception' and an action 'alarm'. The state 'exception' rejects the input 'ticket'. The input 'pass' does, of course, keeps one in the state 'exception'. The 'alarm' can be muted with the input 'mute'. We return to the normal 'locked' state upon the input 'release'.

We will model a language FSML (FSM language) for modeling FSMs. Specifically, we will model several aspects of FSML:

- The visual syntax as used in the figure; see §7.
- A textual syntax as an alternative concrete syntax; see §2.
- An abstract syntax useful for representation in programs; see §3.
- Extra well-formedness constraints on the abstract syntax; see §4.
- A reference semantics for the simulation of FSMs; see §5.
- A code generator translating FSMs into OO programs; see §6.

languages/fsml/sample.fsml

```
initial state locked {
  ticket/collect -> unlocked;
  pass/alarm -> exception;
}
state unlocked {
  ticket/eject;
  pass -> locked;
}
state exception {
  ticket/eject;
  pass;
  mute;
  release -> locked;
}
```

Figure 2: The FSM of Figure 1 in concrete textual syntax.

This primer is concluded by some 'food for thought' in §8. That is, we critically discuss the language model at hand as well as the underlying approach, thereby also motivating reproductions of the model as well as variations on the model, possibly leveraging diverse technologies and techniques.

Various technical details are separated out into the appendix.

2. The concrete textual syntax of FSML

A textual syntax may break down an FSM into 'state declarations', i.e., states with the associated transitions. Each transition identifies the relevant input, the optional action, and the target state. In fact, if the target state is omitted, then this is taken to mean that the transition's target is the current state. See Figure 2 for an illustration.

Let us define the syntax in terms of a context-free grammar. We use an EGL grammar, where EGL stands for 'extended grammar language' and can be regarded as a variation on the Extended Backus Naur Form. EGL is an ad-hoc grammar notation that is provided by the SLEPRO project; see Appendix A for a self-description. In particular, EGL grammars can be executed for the purpose of parsing, but we omit all such details here. See Figure 3 for the grammar. It can be assumed that the concrete textual

languages/fsml/cs.egl

```
fsm : { state }* ;
state : initial 'state' id '{' { transition }* '}' ;
[initial] initial : 'initial' ;
[noninitial] initial : ;
transition : input { '/' action }? { '->' id }? ';' ;
id : name ;
input : name ;
action : name ;
```

Figure 3: Context-free grammar for the concrete syntax of FSML.

syntax, as illustrated in Figure 2, can be parsed indeed with the present grammar.

3. The abstract syntax of FSML

Eventually, we want to manipulate FSMs in (Prolog) programs. To this end, we need a suitable abstract syntax. As an aside, there exist metaprogramming systems that can leverage the concrete syntax directly, without any need for a separate abstract syntax, but we will adopt the simpler approach here to indeed use an abstract syntax. In fact, we assume that abstract syntax trees are (Prolog) terms as in the sense of term algebras or algebraic specifications.

See Figure 4 for an illustration. The abstract syntax describes a FSM as a list of state declarations, which are in turn tuples (triplets) of the form (V, Id, Ts) where V is a Boolean saying whether the state is initial, Id is the assigned id (name) of the state, and Ts is a list of transitions. Each transition is a triplet of the form (In, A, To) where In is the input for the transition, A is the optional action and To is the target state. The optionality of the action is represented such that the missing action is represented by '[]' and the present action, e.g., 'eject', is represented by '[eject]'.

Let us define the abstract syntax in terms of a signature that defines all valid FSM terms. We use an ESL signature, where ESL stands for 'extended signature language' and can be regarded as (quite) a variation on notations familiar from algebraic specification or (less) a variation on type systems used in logic or functional programming. ESL is an ad-hoc signature notation that is provided by the SLEPRO project; see Appendix B for a syntax definition

languages/fsml/sample.term

```
[
  (true,locked,[
   (ticket,[collect],unlocked),
   (pass,[alarm],exception)]),

  (false,unlocked,[
    (ticket,[eject],unlocked),
    (pass,[],locked)]),

  (false,exception,[
    (ticket,[eject],exception),
    (pass,[],exception),
    (mute,[],exception),
    (release,[],locked)])
].
```

Figure 4: The FSM of Figure 1 in abstract syntax.

languages/fsml/as.esl

```
type fsm = state* ;
type state = (initial, id, transition*) ;
type initial = boolean ;
type transition = (input, action?, id) ;
type id = atom ;
type input = atom ;
type action = atom ;
```

Figure 5: Signature for the abstract syntax of FSML.

of the signature notation. In particular, ESL grammars can be executed for conformance checking to see whether a given term conforms to a given signature, but we omit all such details here. See Figure 5 for the signature. The term in Figure 4 conforms indeed to the present signature.

As we intend to parse FSMs via the concrete textual syntax and to manipulate FSMs though via the abstract term-based syntax, we need a mapping from the former to the latter. Of course, both syntaxes are very close to each other in terms of the nonterminals or types defined and in terms of

languages/fsml/cs-to-as.pro

```
% Lexical mapping for name
fsmlMapping(name, String, Atom) :-
  name(Atom, String).

% Mapping for states
fsmlMapping(state, (I1, N, Ts1), (I2, N, Ts2)) :-
  toBoolean((I1==initial), I2),
  map(normalizeTargetState(N), Ts1, Ts2).

% Make target state mandatory
normalizeTargetState(_, (I, A, [S]), (I, A, S)).

% Fill in missing target state as the source state
normalizeTargetState(N, (I, A, []), (I, A, N)).
```

Figure 6: Concrete to abstract syntax mapping for FSML.

the structural breakdown. However, some fine details need to be declared explicitly via a mapping so that parse trees according to the earlier grammar are precisely mapped to terms that conform to the signature for the abstract syntax. The mapping is defined by a Prolog predicate *fsmlMapping*; see the Prolog clauses in Figure 6.

The syntax definition approach of the SLEPRO project assumes such mappings are indeed predicates with three arguments *N*, *T1*, *T2* as follows. *N* is the nonterminal (according to the grammar for the concrete syntax) whose parse trees are to be mapped, i.e., rewritten. *T1* is the (imploded) parse tree that just systematically follows the structure of the grammar. *T2* is the term that should be used in place of *T1*. The starting point is an identity mapping; the mapping predicate only overwrites those cases where the identity mapping is not appropriate. In Figure 6, the first clause replaces the list-of-chars representation of FSM names by proper atoms ('strings'). The second clause replaces 'initial' by 'true' (and 'noninitial' by 'false'); it also fills in the target state, when it is missing, thereby taking care of some syntactic sugar, i.e., the permitted omission of the target state when it equals the source state of a transition.

languages/fsml/ok.pro

```
% Wellness of FSMs
okFsm(Fsm) :-
  require(fsmSingleInitial(Fsm)),
  require(fsmDistinctIds(Fsm)),
  require(fsmResolvable(Fsm)),
  require(fsmDeterministic(Fsm)),
  require(fsmReachable(Fsm)).
```

Figure 7: Overview of constraints imposed on FSML's abstract syntax.

4. Constraints on the abstract syntax

The concrete and abstract syntaxes so far do not cover some of the constraints that we would naturally apply to FSMs. For instance, we did not rule out so far that FSMs have multiple initial states, which is probably not useful. Context-free grammars and signatures are notoriously limited in expressiveness to model such constraints, whereas they are convenient for the basic definition of structure. We need to add extra constraints on top of (say) the abstract syntax definition. In some communities, it is common to actually consider these constraints to form part of the abstract syntax whereas we follow another common view that such constraints are considered separately as in well-formedness or well-typedness judgements used in type systems for programming languages.

See Figure 7 for the Prolog specification listing the named constraints that we consider here. Each constraint gives rise to a separate predicate considered below. Here is short summary. Constraint *fsmSingleInitial* is meant to ensure that there is exactly one initial state declaration. Constraint *fsmDistinctIds* is meant to ensure that the state ids of the state declarations are distinct. Constraint *fsmResolvable* is meant to ensure that all referenced target states are declared. Constraint *fsmDeterministic* is meant to ensure that each given input uniquely defines the target state to be transitioned to, if any. Constraint *fsmReachable* is meant to ensure that all declared states are reachable from the initial state. In Figure 7, all constraint predicates are surround by an application of the meta-predicate *require/1*, which is simply there for better error reporting. That is, if the argument of *require/1* fails, then this is reported immediately, as opposed to simply propagating failure silently.

languages/fsml/ok/initial.pro

```
% There is a single initial state
fsmSingleInitial(Fsm) :-
  findall(
    Initial,
    member((true, Initial, _), Fsm),
    Initials),
  length(Initials, 1).
```

Figure 8: There is only a single state declaration for an initial state.

languages/fsml/tests/initialNotOk.fsml

```
initial state locked {
  ticket/collect -> unlocked;
  pass/alarm -> exception;
}
initial state unlocked {
  ticket/eject;
  pass -> locked;
}
state exception {
  ticket/eject;
  pass;
  mute;
  release -> locked;
}
```

Figure 9: A negative test case: Figure 2 with a second initial state.

All constraints are specified in Figure 8–Figure 18. We also show negative test cases that hence illustrate violation of the constraints; these cases are derived by minimalistic mutation of the sample FSM of Figure 2. The constraints are relatively simple to express using Prolog's meta-predicate *findall/3* which serves here for queries into the terms of the abstract syntax. For instance, in Figure 8, we find all state ids from state declarations with 'true' for the initial component. Once this set is retrieved that the resulting list is of length 1; thus, there is exactly one initial.

The constraint for reachability (Figure 16–Figure 18) is somewhat inter-

languages/fsml/ok/distinct.pro

```
% All state ids are distinct
fsmDistinctIds(Fsm) :-
  findall(
    Id,
    member((_, Id, _), Fsm),
    Ids),
  set(Ids).
```

Figure 10: The state ids of the state declarations are distinct.

languages/fsml/tests/idsNotOk.fsml

```
initial state locked {
  ticket/collect -> unlocked;
  pass/alarm -> exception;
}
state locked {
  ticket/eject;
  pass -> locked;
}
state exception {
  ticket/eject;
  pass;
  mute;
  release -> locked;
}
```

Figure 11: A negative test case: Figure 2 with a double declaration of state 'locked'.

esting, as we have to model a fixed point computation to find all states by repeated consideration of transition as to whether the set of known to be reachable states implies more reachable states. The fixed point criterion is here that we do not find additional states; see Figure 18.

5. The reference semantics for FSML

So far we relies on an intuitive understanding of the FSML semantics, which is reasonable in so far that the notion of finite state machines is pretty fundamental and familiar. Nevertheless, we are bound to provide a precise

languages/fsml/ok/resolvable.pro

```
% All state ids are resolvable to states
fsmResolvable(Fsm) :-
  findall(
    DefId,
    member((_, DefId, _), Fsm),
    DefIds),
  findall(
    RefId,
    (
      member((_, _, Ts), Fsm),
      member((_, _, RefId), Ts)
    ),
    RefIds),
  subset(RefIds, DefIds).
```

Figure 12: All referenced target states are declared.

languages/fsml/tests/resolutionNotOk.fsml

```
initial state locked {
  ticket/collect -> unlockkked;
  pass/alarm -> exception;
}
state unlocked {
  ticket/eject;
  pass -> locked;
}
state exception {
  ticket/eject;
  pass;
  mute;
  release -> locked;
}
```

Figure 13: A negative test case: Figure 2 with a unresolvable state reference 'unlockkked'.

and executable reference semantics of the language which can then be hold accountable by other language-based software components, e.g., a code generator (see next section) or a refactoring.

languages/fsml/ok/deterministic.pro

```
% Input is handled deterministically
fsmDeterministic(Fsm) :-
  map(stateDeterministic, Fsm).

% Input is handled deterministically in a state
stateDeterministic((_, _, Ts)) :-
  findall(
    I,
    member((I, _, _), Ts),
    Is),
  set(Is).
```

Figure 14: Each given input uniquely defines the target state to be transitioned to, if any.

languages/fsml/tests/determinismNotOk.fsml

```
initial state locked {
  ticket/collect -> unlocked;
  ticket/alarm -> exception;
}
state unlocked {
  ticket/eject;
  pass -> locked;
}
state exception {
  ticket/eject;
  pass;
  mute;
  release -> locked;
}
```

Figure 15: A negative test case: Figure 2 with a nondeterministic transition; see the double occurrence of 'ticket' in state 'locked'.

Let us set up a test case for the FSM of Figure 2. That is, a valid input sequence for the FSM is shown in Figure 19 and the corresponding output in terms of actions and state changes is shown in Figure 19. We shall define a semantics that indeed transforms the input into the output.

We speak of a reference semantics here to emphasize that this semantics

languages/fsml/ok/reachable.pro

```
% All states are reachable from the initial state
fsmReachable(Fsm) :-
  findall(
    Id,
    member((_, Id, _), Fsm),
    Ids),
  findall(
    Initial,
    member((true, Initial, _), Fsm),
    Initials),
  fixedPointIds(Fsm, Initials, Reachables),
  setEq(Ids, Reachables).
```

Figure 16: All declared states are reachable from the initial state.

languages/fsml/tests/reachabilityNotOk.fsml

```
initial state locked {
  ticket/collect -> locked;
  pass/alarm -> exception;
}
state unlocked {
  ticket/eject;
  pass -> locked;
}
state exception {
  ticket/eject;
  pass;
  mute;
  release -> locked;
}
```

Figure 17: A negative test case: Figure 2 with an unreachable state ‘unlocked’.

may not be practically useful immediately. That is, we will define the semantics of a FSM by a simulation judgement which assumes that all input is provided upfront and all output (the action sequence) is provided upon completion of input processing. This is a batch model, which is useful in a formal semantics, but it is not useful for using FSMs in interactive systems.

languages/fsml/ok/closure.pro

```
% Compute closure of state reachable
fixedPointIds(Fsm, Ids1, Ids4) :-
  findall(
    Id2,
    (
      member(Id1, Ids1),
      member((_, Id1, Ts), Fsm),
      member((_, _, Id2), Ts)
    ),
    Ids2),
  union(Ids1, Ids2, Ids3),
  ( \+ setEq(Ids1, Ids3) ->
        fixedPointIds(Fsm, Ids3, Ids4)
      ; Ids4 = Ids1
  ).
```

Figure 18: Fixed point computation needed in Figure 16.

languages/fsml/sample.input

```
[
  ticket, % Regular insertion of a ticket
  pass, % Regular passage of turnstile
  ticket, % Regular insertion of a ticket
  pass, % Regular passage of turnstile
  ticket, % Regular insertion of a ticket
  ticket, % Tickets are ejected in unlocked state
  pass, % Regular passage of turnstile
  pass, % Attempt leads to exceptional state
  ticket, % Tickets are ejected in exceptional state
  pass, % Passage attempt keeps us in exceptional state
  mute, % Mute indeed
  release, % Return to normal
  ticket, % Regular insertion of a ticket
  pass % Regular passage of turnstile
].
```

Figure 19: A possible input sequence for FSM of Figure 2.

languages/fsml/sample.output

```
[
  ([collect], unlocked),
  ([], locked),
  ([collect], unlocked),
  ([], locked),
  ([collect], unlocked),
  ([eject], unlocked),
  ([], locked),
  ([alarm], exception),
  ([eject], exception),
  ([], exception),
  ([], exception),
  ([], locked),
  ([collect], unlocked),
  ([], locked)
].
```

Figure 20: The output for Figure 19.

See Figure 21 for the semantics; the judgements are expressed in Prolog. The clause at the top selects the initial state of the FSM and enters the simulation with that state and the complete input. The remaining two clauses describe the iteration over the input such that a suitable transition is looked up in each step and action (if any) and the new state are paired up as an element in the output so that simulation proceeds with the new state and the remaining input.

6. The code generator for FSML

A practically useful interpretation of FSML would be achieved, if we enabled interactive execution of the FSMs in the context of some programming environment. We pick Java as our target here. We describe a code generator which translates FSMs into Java code. The generated code can be augmented with hand-written components specifically for the intended semantics of actions. The Java representation of FSMs meets the requirement that execution is step-by-step. That is, a transition is transparently realized including the potential performance of an action whenever an input item arrives.

languages/fsml/simulation.pro

```
% Simulate FSM for given input to compute output
simulateFsm(
  Fsm,
  Input,
  Output
) :-
      member((true, Id, _), Fsm), % Select initial state
      simulateFsm_(Fsm, Id, Input, Output).

% All input was processed
simulateFsm_(_, _, [], []).

% Apply transition for input at hand
simulateFsm_(Fsm, Id1, [Input|Inputs], [(Action, Id2)|Outputs]) :-
  member((_, Id1, Ts), Fsm),
  member((Input, Action, Id2), Ts),
  simulateFsm_(Fsm, Id2, Inputs, Outputs).
```

Figure 21: Semantics of FSML.

languages/fsml/java/State.java

```
// Generated code
public enum State {locked, unlocked, exception}
```

Figure 22: Java representation of the states for the turnstile example.

There are many different ways to generate imperative/OO code for FSMs. We pick one particular option here–without loss of generality. The chosen option can be said to be data-centric in that FSMs are represented in an appropriate container-based data structure, whereas the execution of FSMs relies on the interpretation of said data structure. Further, the programmer-defined meaning of actions is provided by a handler which interpreters action names.

Figure 22–Figure 24 shows (generated) Java enum types for the states, inputs, and actions. Obviously, these sets are trivially obtainable from the abstract syntactical representation of an FSM. In Figure 25, this is illustrated for the enum type for states. The predicate collects all (declared) state ids

languages/fsml/java/Input.java

```
// Generated code
public enum Input {ticket, pass, mute, release}
```

Figure 23: Java representation of the inputs for the turnstile example.

languages/fsml/java/Action.java

```
// Generated code
public enum Action {collect, alarm, eject}
```

Figure 24: Java representation of the actions for the turnstile example.

languages/fsml/to-java/state.pro

```
statesOfFsm(Fsm, Text) :-

  % Collect declared states
  findall(
    Id,
    member((_, Id, _), Fsm),
    Ids),

  % Render enum type
  ppJavaDecl(enum(public, 'State', Ids), Text).
```

Figure 25: Code generation for state type.

and assembles an enum type, subject to a suitable abstract syntax of Java; see Appendix C for a summary of the (abstract) Java syntax used by the code generator. Finally, the Java declaration is pretty printed; see the use of *ppJavaDecl/2* in the figure, subject to a pretty printer for the Java language. The generator predicates for the enum types for inputs and actions are quite similar.

Figure 26 shows a generic, i.e., FSM-independent interface for defining handlers for FSM actions. Figure 27 shows a trivial implementation of said interface for the turnstile example. The actions simply print out the names of the actions, but it is clear that arbitrary functionality can be plugged into FSM execution in this manner.

Figure 28 shows essentially the runtime for FSMs. This is generic code

languages/fsml/java/HandlerBase.java

```
// Reusable code
public interface HandlerBase<A> {
    public void handle(A a);
}
```

Figure 26: Generic interface for action handers.

languages/fsml/java/Handler.java

```
// Turnstile-specific, handwritten code
public class Handler implements HandlerBase<Action> {
    public void handle(Action a) {
        switch (a) {
        case collect :
            System.out.println("collect");
            break;
        case alarm :
            System.out.println("alarm");
            break;
        case eject :
            System.out.println("eject");
            break;
        }
    }
}
```

Figure 27: A trivial interpretation of the turnstile actions.

which is appropriately parameterized in the types for states, inputs, and actions. The class relies on a table for holding the transitions in the form of a map from states to maps from inputs to pairs of optional actions and states. There is an appropriate *add* method which makes it easy to insert into this non-basic container. The class is also prepared to keep track of and make use of a handler for the involved actions. The class is called *StepperBase* because its key method is the one for performing a step (a transition) based on an actual input item.

Figure 29 shows the generated code for the turnstile-specific subclass of the *StepperBase* class. The generic parameters of *StepperBase* are instan-

languages/fsml/java/StepperBase.java

```
import java.util.HashMap;

// Reusable code
public abstract class StepperBase<S, I, A> {
    protected S state;
    protected HandlerBase<A> handler;
    private HashMap<S,HashMap<I,Pair<A,S>>> table =
        new HashMap<S,HashMap<I,Pair<A,S>>>();
    public final void add(S from, I i, A a, S to) {
        if (!table.containsKey(from))
            table.put(from, new HashMap<I,Pair<A,S>>());
        HashMap<I,Pair<A,S>> subtable = table.get(from);
        Pair<A,S> pair = new Pair<A,S>(a, to);
        subtable.put(i, pair);
    }
    public final void step(I i) {
        HashMap<I,Pair<A,S>> subtable = table.get(state);
        Pair<A,S> pair = subtable.get(i);
        S from = state;
        S to = pair.y;
        System.out.println("from: "+from+", input: "+i+", to: "+to);
        if (pair.x!=null) handler.handle(pair.x);
        state = to;
    }
}
```

Figure 28: Generic class for FSM execution.

tiated to the turnstile-specific enum types for states, inputs, and actions, which were shown earlier. Also, the class encodes all transitions of the turnstile FSM via appropriate calls to the *add* method, as part of the constructor. The initial state is also set up and the turnstile-specific handler is communicated as well.

Figure 30 shows the code generation predicate for steppers. The predicate refers to more elements of the Java language: several expression forms (e.g., 'null'), assignment statements, method calls, and class declarations. In a final step, the constructed abstract syntactical representation is passed to the pretty printer for Java.

languages/fsml/java/Stepper.java

```
// Generated code
public class Stepper extends StepperBase<State, Input, Action> {
  public Stepper(HandlerBase<Action> handler) {
    this.handler = handler;
    state = State.locked;
    add(State.locked, Input.ticket, Action.collect, State.unlocked);
    add(State.locked, Input.pass, Action.alarm, State.exception);
    add(State.unlocked, Input.ticket, Action.eject, State.unlocked);
    add(State.unlocked, Input.pass, null, State.locked);
    add(State.exception, Input.ticket, Action.eject, State.exception);
    add(State.exception, Input.pass, null, State.exception);
    add(State.exception, Input.mute, null, State.exception);
    add(State.exception, Input.release, null, State.locked);
  }
}
```

Figure 29: Configuration of a stepper for the turnstile example.

7. The visual syntax of FSML

It remains to define the visual syntax of FSML, as it was illustrated in Figure 1. A visual syntax definition may serve these major purposes: actual representation (visualization) and editing. Indeed, we want this syntax to be complemented by rendering support. We do not consider editing here.

The visualization in Figure 1 renders an FSM as a graph. There are nodes and edges; both of them can be labeled; edges are directed; nodes are of a certain shape (an ellipse here) and style (regular or filled). **??** represents represents the graph in the abstract (Prolog-based) syntax of a simple graph description based on the DGL notation provided by the SLEPRO project. DGL stands for 'dot-based graph language'; see Appendix D for this notation. Nodes are quadruplets of the form (*Id*, *Label*, *Shape*, *Style*); edges are triplets of the form (*FromId*, *ToId*, *OptionalLabel*).

The DGL processor translates the graph into the concrete syntax of the *dot* language of *GraphViz*[1]; see **??**. (That is, DGL can be viewed as a subset of the DOT language.) The GraphViz tool directly renders the dot

[1] `http://www.graphviz.org/`

languages/fsml/to-java/stepper.pro

```
stepperOfFsm(Fsm, Text) :-

  % Map transitions to calls to the stepper's "add" method
  findall(S,
    (
      % Iterate over states and transitions
      member((_, From, Ts), Fsm),
      member((I, A, To), Ts),

      EFrom = select(name('State'), From), % Expression for source state
      ETo = select(name('State'), To), % Expression for target state
      EI = select(name('Input'), I), % Expression for input

      % Expression for (optional) action
      (A = [A1] -> AE = select(name('Action'), A1); AE = null),

      % Expression statement for "add"
      S = expression(call(add, [EFrom, EI, AE, ETo]))
    ),
    Ss),

  HA = assign(select(this, handler), name(handler)), % Handler assignment
  SI = assign(name(state), select(name('State'),locked)), % State initialization

  % Stepper subclass
  Class = class(public, false, 'Stepper', [],
    [typeapp('StepperBase',
      [typename('State'), typename('Input'), typename('Action')])],
    [constr(
      public,
      [(typeapp('HandlerBase', [typename('Action')]), handler)],
      [HA,SI|Ss])]),

  % Render the class
  ppJavaDecl(Class, Text).
```

Figure 30: Code generation for stepper class.

languages/fsml/dot/sample.term

```
(
  % States
  [
    (locked,locked,ellipse,[filled]),
    (unlocked,unlocked,ellipse,[]),
    (exception,exception,ellipse,[])
  ],
  % Edges
  [
    (locked,unlocked,['ticket/collect']),
    (locked,exception,['pass/alarm']),
    (unlocked,unlocked,['ticket/eject']),
    (unlocked,locked,[pass]),
    (exception,exception,['ticket/eject']),
    (exception,exception,[pass]),
    (exception,exception,[mute]),
    (exception,locked,[release])
  ]
).
```

Figure 31: The FSM of Figure 1 in the abstract syntax of the DGL language.

specification as shown Figure 1.

8. Food for thought

Let us discuss some characteristics, limitations, or opportunities for alternative or additional components. In this manner, we prepare for a good number of experiments that could be carried out—either to learn more about DSLs or to demonstrate other technologies and techniques:

Leverage a programming ecosystem The FSML model of this document directly relies on Prolog or language modeling notations that are in turn implemented in Prolog, as part of the SlePro project. Alternatively, one could model (implement) FSML also in some existing programming ecosystem, e.g., in Java, Ruby, or Python, also subject to the use of appropriate libraries, parser generators, and other tools. This

languages/fsml/dot/sample.dgl

```
digraph G {
  locked [label="locked", shape=ellipse, style=filled]
  unlocked [label="unlocked", shape=ellipse]
  exception [label="exception", shape=ellipse]
  locked -> unlocked [label=" ticket/collect "]
  locked -> exception [label=" pass/alarm "]
  unlocked -> unlocked [label=" ticket/eject "]
  unlocked -> locked [label=" pass "]
  exception -> exception [label=" ticket/eject "]
  exception -> exception [label=" pass "]
  exception -> exception [label=" mute "]
  exception -> locked [label=" release "]
}
```

Figure 32: The FSM of Figure 1 in the dot language.

may enable interesting comparison across programming languages and ecosystems.

Leverage a language workbench Language modeling is readily addressed by so-called language workbenches such as Rascal or Spoofax. Thus, it would only be natural to exercise such workbenches for the present DSL scenario. In fact, the overall domain of FSMs is generally a popular one, when it comes DSL modeling (implementation). Thus, some similar model may readily exist for some of these workbenches.

Leverage model-driven engineering Another existing approach to language modeling is based on model-driven engineering (MDE). For instance, one could use EMF and GMF for language modeling, thereby covering abstract and visual syntax for graphical editors. The actual functionality (such as constraints, semantics, and code generator) could leverage a programming language which readily collaborates with the chosen MDE approach, e.g., Java in the case of EMF/GMF. Some of this functionality could also be expressed with model transformations in designated languages, e.g., ATL or QVT. Further, specific model-to-text and text-to-model technologies may be leveraged in the context of MDE.

Concrete object syntax The FSML model of the present document leverages abstract syntax for all the notations that need to be manipulated: FSML, Java, and DGL. Some metaprogramming systems and language workbenches also support concrete object syntax, e.g., Stratego and TXL. In this manner, less syntaxes need to be defined, metaprograms remain closer to the notation that one may be used for the manipulated artifacts. In the future, support for concrete object syntax may be added to SLEPRO.

Template-based code generation The FSML model underlying the present document leverages pretty-printing combinators for mapping abstract to concrete syntax. In fact, these mapping components are not shown in the document because they are concerned with languages other than FSML, namely Java and DGL. Nevertheless, it may be worthwhile to investigate other options for mapping abstract to concrete syntax—in particular, template-based code generation or model-to-text transformations, as mentioned earlier in the context of MDE.

Alternative code generation schemes The code generator favors one particular scheme such that FSMs are essentially represented as data and actions are to be plugged into FSM execution through appropriate handlers. Several other schemes for the OO representation of FSMs are known. For instance, transitions may also be represented as control-flow code, i.e., by using if or switch statements dispatching on states and inputs. Alternatively, states may also be thought of as objects with transitions modeled as polymorphic methods. Without caring much here about the pros and cons of these approaches, we could simply want to study them from the technical point of view of code generation. In fact, we picked the data-centric approach, as it was obvious that code generation is straightforward in this case.

An editor for FSML Language support may also include editing support as in the sense of a structure editor for concrete textual syntax or a graphical editor for concrete visual syntax (perhaps based on GMF, as mentioned earlier).

A proper FSML-based application The value of the FSML language may be more obvious, if we managed to demonstrate it in an actual application such as a concrete turnstile system. One could be looking at

embedded systems scenarios, for example.

languages/egl/cs.egl

```
grammar : {rule}∗ ;
rule : {'[' label ']'}? nonterminal ':' symbols ';' ;
[t] symbol : terminal ;
[n] symbol : nonterminal ;
[star] symbol : '{' symbols '}' '∗' ;
[plus] symbol : '{' symbols '}' '+' ;
[option] symbol : '{' symbols '}' '?' ;
symbols : {symbol}∗ ;
label : name ;
terminal : qstring ;
nonterminal : name ;
```

Figure A.33: EGL grammar of EGL grammars.

Appendix A. Grammar of EGL grammars

EGL was used for the concrete syntax definition of FSML in §2. EGL is an EBNF-like notation. See Figure A.33 for a self-description of the grammar notation.

languages/esl/cs.egl

```
signature : { decl ';' }* ;
[type] decl : 'type' name '=' typeexpr ;
[symbol] decl : 'symbol' name ':' args '->' name ;
args : { typeexpr { 'x' typeexpr }* }? ;
typeexpr : factor cardinality ;
[term] factor : 'term' ;
[atom] factor : 'atom' ;
[integer] factor : 'integer' ;
[float] factor : 'float' ;
[number] factor : 'number' ;
[boolean] factor : 'boolean' ;
[tuple] factor : '(' typeexpr {',' typeexpr}+ ')' ;
[sort] factor : name ;
[star] cardinality : '*' cardinality ;
[plus] cardinality : '+' cardinality ;
[option] cardinality : '?' cardinality ;
[none] cardinality : ;
```

Figure B.34: EGL grammar of ESL signatures.

Appendix B. Grammar of ESL signature

ESL was used for the abstract syntax definition of FSML in §3. ESL is a type-declaration notation inspired by algebraic signatures, term algebras, and algebraic data types. See Figure B.34 for the concrete syntax of the notation. ESL is also used in some of the appendix sections that follow.

languages/java/as.esl

```
symbol class : visible x abstract x name x tpara* x extends x member* -> decl ;
symbol enum : visible x name x name+ -> decl ;
symbol public : -> visible ;
symbol protected : -> visible ;
symbol private : -> visible ;
type abstract = boolean ;
type name = atom ;
type tpara = name ;
type extends = type? ;
symbol constr : visible x mpara* x statement* -> member ;
symbol method : visible x time x type x name x mpara* x statement* -> member ;
symbol assignment : expression x expression -> statement ;
symbol expression : expression -> statement ;
symbol this : -> expression ;
symbol null : -> expression ;
symbol name : name -> expression ;
symbol select : expression x name -> expression ;
symbol call : name x expression* -> expression ;
type mpara = (type, name) ;
symbol typename : name -> type ;
symbol typeapp : name x type* -> type ;
```

Figure C.35: ESL signature for the abstract syntax of a Java subset.

Appendix C. Abstract syntax of a Java subset

See Figure C.35. The subset is assumed by the code generator of §6.

languages/dgl/as.esl

```
type graph = (node*, edge*) ;
type node = (id, label, shape, style?) ;
type edge = (id, id, label?) ;
type id = atom ;
type label = atom ;
symbol box : -> shape ;
symbol ellipse : -> shape ;
symbol bold : -> style ;
symbol dotted : -> style ;
symbol filled : -> style ;
```

Figure D.36: ESL signature for the abstract syntax of DGL.

Appendix D. Abstract syntax of DGL

We used DGL in §7 to export FSMs as graphs in a manner that they can be visualized. DGL is a simple graph description language. DGL stands 'dot-based graph language' to hint at the fact that the abstract syntax of DGL can be (is) rendered in the dot language of Graphviz. The abstract syntax of DGL is shown in Figure D.36.

languages/fsml/tests/parserError.fsml

```
innnitial state locked {
  ticket/collect -> unlocked;
  pass/alarm -> exception;
}
state unlocked {
  ticket/eject;
  pass -> locked;
}
state exception {
  ticket/eject;
  pass;
  mute;
  release -> locked;
}
```

Figure D.37: A negative test case: Figure 2 with a syntax error; see 'innnitial'.

languages/fsml/tests/illegalSymbol.input

```
[
  foo % This is a not a valid input symbol
].
```

Figure D.38: A negative test case: an input for Figure 2 which contains an invalid input symbol.

languages/fsml/tests/infeasibleSymbol.input

```
[
  mute % This symbol is only feasible in the exceptional state
].
```

Figure D.39: A negative test case: an input for Figure 2 which contains an infeasible input symbol in the pool position—the initial state 'locked' does not accept the 'mute' input.

Appendix E. Additional test cases for the specification

Let us add a few more negative test cases in addition to the illustrations of constraint violation of §4. A simple parsing error is caused by the input of Figure D.37. Figure D.38 and Figure D.39 demonstrate two major scenarios

for failure of ‘simulation’ according to §5.