Chapter 2
A Story of a Domain-Specific Language

MARTIN FOWLER.[1]

Abstract In this chapter, several fundamental concepts and engineering techniques for software languages are explained by means of an illustrative domain-specific language. In particular, we exercise the internal and external styles of DSL implementation, textual and visual syntax, parsing, interpretation, and code generation. As a running example, we deal with a DSL for finite state machines FSML (*FSM L*anguage). In addition to implementing FSML with mainstream languages and technologies, we discuss design and implementation options and concerns overall and we describe a number of "recipes" for DSL development.

[1] There is no "Greek" in Martin Fowler's textbooks on refactoring [4] and DSLs [5], both addressing important topics in software language engineering. These accessible textbooks triggered research on these topics and connected research better with "mainstream" software development. Martin Fowler was again visionary when he asked in 2005 "Language Workbenches: The Killer-App for Domain Specific Languages?" (`https://www.martinfowler.com/articles/languageWorkbench.html`), thereby fueling the development of and research on language workbenches [2, 3, 9, 8, 16, 15, 17, 13].

© Springer International Publishing AG, part of Springer Nature 2018
R. Lämmel, *Software Languages*,
https://doi.org/10.1007/978-3-319-90800-7_2

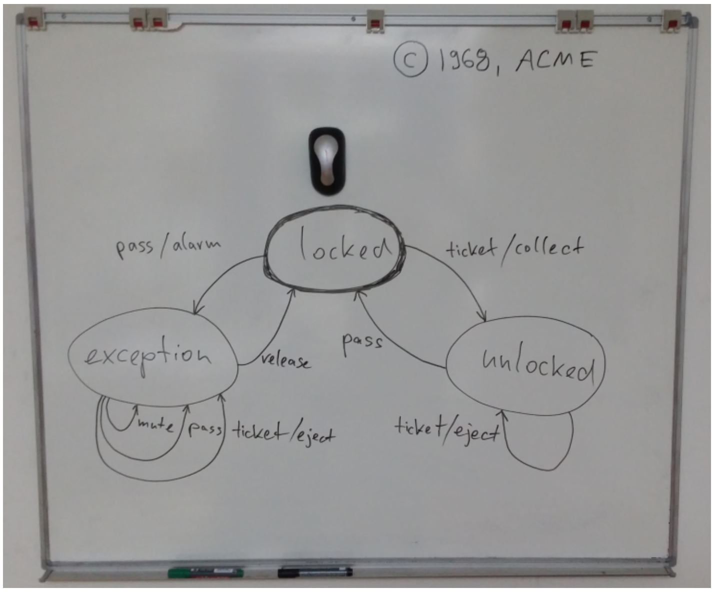

Fig. 2.1 A turnstile FSM in visual notation on the whiteboard.

2.1 Language Concepts

We assume an imaginary business context: a company *Acme*[2], which develops *embedded systems*.[3] *Acme* is an international leader in embedded systems development. Over the last 50 years, *Acme* has matured a number of development techniques that are specifically tailored to the area of embedded systems development. For instance, *Acme* uses FSMs[4] in the development of embedded systems. In this chapter, we discuss a corresponding language for FSMs, FSML (*FSM L*anguage). FSML is a domain-specific language that was grown at *Acme* over many years; the language and its implementation have emerged and evolved in various ways, as described below.

FSML is introduced here by means of an example. In an ongoing project, *Acme* is developing a turnstile component, as part of a bigger contract to modernize the metro system in an undisclosed city. Metro passengers need to pass through the turnstile (a hub or spider) and insert a valid ticket into the turnstile's card reader when they want to reach the platform in a legal manner. The *Acme* architects and the customer agree on the basic functionality for turnstiles in a meeting, where they draw an FSM on the whiteboard as shown in Fig. 2.1.

2 http://en.wikipedia.org/wiki/Acme_Corporation
3 http://en.wikipedia.org/wiki/Embedded_system
4 http://en.wikipedia.org/wiki/Finite-state_machine

FSML is quickly explained in terms of its visual notation with the example at hand. FSMs comprise states (nodes) and transitions (directed edges). The initial state of the machine is indicated by a bolder border. These are these states in the turnstile FSM:

- *locked*: The turnstile is locked. No passenger is allowed to pass.
- *unlocked*: The turnstile is unlocked. A passenger may pass.
- *exception*: A problem has occurred and metro personnel need to intervene.

Each transition connects two states and is annotated by two parts, e/a, an event e and an action a, where the latter is optional. The event may be triggered by the user; this may involve sensors in an embedded system. An event causes a transition. An action corresponds to functionality to be performed upon a transition; this may involve actors in an embedded system. The source of a transition is the source state; the target of a transition is the target state. The turnstile FSM involves these *events*:

- *ticket*: A passenger inserts a ticket into the card reader.
- *pass*: A passenger passes through the turnstile as noticed by a sensor.
- *mute*: Metro personnel turn off the alarm after an exception.
- *release*: Metro personnel turn on normal operation again.

The turnstile FSM involves these *actions*:

- *collect*: The ticket is collected by the card reader.
- *eject*: The ticket is ejected by the card reader.
- *alarm*: An alarm is turned on, thereby requesting metro personnel.

Based on such an understanding of states, events, and actions, the meaning of the different *transitions* in Fig. 2.1 should be obvious by now. Consider, for example, the transition from the source state "locked" to the target state 'unlocked', which is annotated by "ticket/collect" to mean that the transition is triggered by the event of inserting a ticket and the transition causes the action of collecting the ticket.

The idea is now that architects and customers can validate their intuitions about turnstiles by starting from some input (a sequence of events) and determine the corresponding output (a sequence of actions), as illustrated below.

Illustration 2.1 (Sample input for the turnstile FSM)
The input is a sequence of the following events:

ticket	*A ticket is inserted. (The turnstile is thus unlocked.)*
ticket	*Another ticket is inserted. (The superfluous ticket is ejected.)*
pass	*Someone passes through the turnstile. (This is OK.)*
pass	*Someone else passes through the turnstile. (This triggers an alarm.)*
ticket	*A ticket is inserted. (The ticket is ejected in the exceptional state.)*
mute	*The alarm is muted.*
release	*Metro personnel switch back to normal.*

Illustration 2.2 (Sample output for the sample input of Illustration 2.1)
The output is a sequence of the following actions:

collect	*The inserted ticket is collected.*
eject	*A ticket inserted in the unlocked state is ejected.*
alarm	*An attempt to pass in the locked state triggers an alarm.*
eject	*A ticket inserted in the exceptional state is ejected.*

2.2 Internal DSL

Over the years, the *Acme* engineers increasingly appreciated the FSM notation. There was growing interest in handling FSMs as proper software engineering artifacts, as opposed to simply passing down whiteboard drawings from architects to developers.

DSL implementation efforts were sparked off within the company. One engineer implemented FSML as an *internal DSL* [5, 14, 1, 12] in Java. In this manner, a machine-checked and executable notation for FSMs was obtained without much effort, and also without the need for special tools.

In the internal style of DSL implementation, DSL programs are represented and their behavior is implemented in a host language. The idea is that the language concepts of the DSL are implemented as a library and the DSL programmer is provided with an API for manipulating DSL programs. We demonstrate the use of Java and Python as host languages for FSML here. We should mention that the details of internal DSL style depend significantly on the host language. If we were using C++, Scheme, Haskell, Scala, or some other language as the host language, additional or different techniques could be leveraged, for example, operator overloading, macros, or templates.

2.2.1 Baseline Object Model

Let us begin with a very simple object model for FSML. We assume classes Fsm, State, and Transition for the representation of FSMs, states, and transitions. Setter/getter-based Java code for FSM construction may take the form as shown below.

Note: Most “code snippets” (from Chapter 2 onwards) in this book are enclosed into “**Illustration**” blocks and contain a clickable URL so that the corresponding source file can be looked up in the underlying online repository. Many source files are shown with elisions and, thus, by following the link, one can inspect the complete file and also observe the context of the file in the repository. The following illustration contains the URL above the actual source code; see the underlined string.

Illustration 2.3 (Imperative style of constructing FSML objects)

Java source code org/softlang/fsml/ImperativeSample.java

```java
turnstile = new Fsm();
State s = new State();
s.setStateid("locked");
s.setInitial(true);
turnstile.getStates().add(s);
s = new State();
s.setStateid("unlocked");
turnstile.getStates().add(s);
s = new State();
s.setStateid("exception");
turnstile.getStates().add(s);
Transition t = new Transition();
t.setSource("locked");
t.setEvent("ticket");
t.setAction("collect");
t.setTarget("unlocked");
turnstile.getTransitions().add(t);
t = new Transition();
... // add more transitions
```

That is, various objects are to be constructed and initialized with setters and other accessors. This style may be slightly improved if *functional constructors* are put to work as shown below.

Illustration 2.4 (Functional construction of FSML objects)

Java source code org/softlang/fsml/FunctionalSample.java

```java
turnstile = new Fsm();
turnstile.getStates().add(new State("locked", true));
turnstile.getStates().add(new State("unlocked"));
turnstile.getStates().add(new State("exception"));
turnstile.getTransitions().add(new Transition("locked", "ticket", "collect", "unlocked"));
turnstile.getTransitions().add(new Transition("locked", "pass", "alarm", "exception"));
... // add more transitions
```

The code is still littered with object construction, container manipulation, and the repetition of source states for transitions. We will discuss below a more “fluent” API. As a baseline, we implement a simple baseline as follows.

Illustration 2.5 (Object model with functional constructors)

Java source code org/softlang/fsml/Fsm.java

```
public class Fsm {
  private List<State> states = new LinkedList<>();
  private List<Transition> transitions = new LinkedList<>();
  public List<State> getStates() { return states; }
  public List<Transition> getTransitions() { return transitions; }
}
```

Java source code org/softlang/fsml/State.java

```
public class State {
  private String id;
  private boolean initial;
  public String getStateid() { return id; }
  public void setStateid(String state) { this.id = state; }
  public boolean isInitial() { return initial; }
  public void setInitial(boolean initial) { this.initial = initial; }
  public State() { }
  public State(String id) { this.id = id; }
  public State(String id, boolean initial) { this.id = id; this.initial = initial; }
}
```

Java source code org/softlang/fsml/Transition.java

```
public class Transition {
  private String source;
  private String event;
  private String action;
  private String target;
  ... // getters and setters omitted
  public Transition() { }
  public Transition(String source, String event, String action, String target) {
    this.source = source;
    this.event = event;
    this.action = action;
    this.target = target;
  }
}
```

Exercise 2.1 (Object model with references) [Basic level]
Implement an alternative object model where the target state of a transition is modeled as a proper object reference to a state object, as opposed to the use of strings in the baseline model.

2.2.2 Fluent API

Let us aim at a more “fluent” API, which is focused on language concepts, eliminates sources of redundancy, and hides the object-oriented representation, as shown for the host languages Java and Python below.

Illustration 2.6 (Fluent style of representing an FSM in Java)

Java source code org/softlang/fsml/fluent/Sample.java

```
turnstile = fsm()
  .addState("locked")
    .addTransition("ticket", "collect", "unlocked")
    .addTransition("pass", "alarm", "exception")
  .addState("unlocked")
    .addTransition("ticket", "eject", "unlocked")
    .addTransition("pass", null, "locked")
  .addState("exception")
    .addTransition("ticket", "eject", "exception")
    .addTransition("pass", null, "exception")
    .addTransition("mute", null, "exception")
    .addTransition("release", null, "locked");
```

Illustration 2.7 (Fluent style of representing an FSM in Python)

Python module FsmlSample

```
turnstile = \
  Fsm() \
    .addState("locked") \
      .addTransition("ticket", "collect", "unlocked") \
      .addTransition("pass", "alarm", "exception") \
    .addState("unlocked") \
      .addTransition("ticket", "eject", "unlocked") \
      .addTransition("pass", None, "locked") \
    .addState("exception") \
      .addTransition("ticket", "eject", "exception") \
      .addTransition("pass", None, "exception") \
      .addTransition("mute", None, "exception") \
      .addTransition("release", None, "locked")
```

The construction of the FSM is expressed in a relatively concise and readable manner. The choice of Java or Python as a host language does not influence the notation much. The fluent API style is achieved by applying a few simple techniques:

Factory methods Rather than invoking regular constructors, any sort of DSL program fragment is constructed by appropriate factory methods. In this manner, we effectively abstract from the low-level representation of DSL programs. Also, the DSL concepts map more systematically to API members and the verbosity of constructor invocation is avoided.

Method chaining A DSL program is represented as a chain of object mutations such that each step returns a suitable object on which to perform the next step. For the simple DSL at hand, the returned object is the FSM to add transitions. In this manner, DSL programs can be represented as expressions instead of statement sequences on local variables.

Implicit parameters The API for DSL program construction may maintain implicit parameters so that they do not need to be repeated explicitly. For FSML, it is natural to group all transitions by source state and, thus, the API maintains a "current" state.

Conventions (defaults) Some details may be omitted by the programmer if reasonable defaults can be assumed, subject to conventions. For FSML, it makes sense to assume that the first state is the initial state and, thus, the flag "initial" can be omitted universally.

Let us illustrate the fluent API in Java.

Illustration 2.8 (A fluent Java API for FSMs)

Java source code org/softlang/fsml/fluent/Fsm.java

```java
public interface Fsm {
  public Fsm addState(String state);
  public Fsm addTransition(String event, String action, String target);
  public String getInitial();
  public ActionStatePair makeTransition(String state, String event);
}
```

Java source code org/softlang/fsml/fluent/ActionStatePair.java

```java
// Helper class for "makeTransition"
public class ActionStatePair {
  public String action;
  public String state;
}
```

The API does not just feature members for construction; it also provides access to the initial state and the transitions, thereby preparing for the 'interpretation' of FSMs, as discussed later in detail (Section 2.2.3). Let us illustrate one option of implementing the fluent API such that we use a cascaded map to maintain states and transitions as shown below.

Illustration 2.9 (A fluent API implementation for FSML in Java)

Java source code org/softlang/fsml/fluent/FsmImpl.java

```
public class FsmImpl implements Fsm {
  private String initial; // the initial state
  private String current; // the "current" state
  // A cascaded map for maintaining states and transitions
  private HashMap<String, HashMap<String, ActionStatePair>> fsm =
      new HashMap<>();
  private FsmImpl() { }
  // Construct FSM object
  public static Fsm fsm() { return new FsmImpl(); }
  // Add state and set it as current state
  public Fsm addState(String id) {
    // First state is initial state
    if (initial == null) initial = id;
    // Remember state for subsequent transitions
    this.current = id;
    if (fsm.containsKey(id)) throw new FsmlDistinctIdsException();
    fsm.put(id, new HashMap<String, ActionStatePair>());
    return this;
  }
  // Add transition for current state
  public Fsm addTransition(String event, String action, String target) {
    if (fsm.get(current).containsKey(event)) throw new FsmlDeterminismException();
    ActionStatePair pair = new ActionStatePair();
    pair.action = action;
    pair.state = target;
    fsm.get(current).put(event, pair);
    return this;
  }
  // Getter for initial state
  public String getInitial() {
    return initial;
  }
  // Make transition
  public ActionStatePair makeTransition(String state, String event) {
    if (!fsm.containsKey(state)) throw new FsmlResolutionException();
    if (!fsm.get(state).containsKey(event)) throw new FsmlInfeasibleEventException();
    return fsm.get(state).get(event);
  }
}
```

The implementation makes the assumption that the first state corresponds to the initial state. Also, when a transition is added, the most recently added state (current) serves as the source state. The implementation also shields against some programming errors when describing an FSM; see the exceptions raised. We will discuss the related constraints later on (Section 2.2.4).

Exercise 2.2 (Fluent API on baseline object model) [Basic level]
Provide an alternative implementation of the fluent API such that the API is realized on top of the baseline object model of Section 2.2.1.

We also exercise Python as the host language for API implementation as follows.

Illustration 2.10 (A fluent API implementation for FSML in Python)

Python module FsmlModel

```
class Fsm():
    def __init__(self):
      self.fsm = defaultdict(list)
      self.current = None
    def addState(self, id):
        return self.addStateNoDefault(self.current is None, id)
    def addStateNoDefault(self, initial, id):
        if id in self.fsm[id]: raise FsmlDistinctIdsException;
        self.stateObject = dict()
        self.stateObject['transitions'] = defaultdict(list)
        self.stateObject['initial'] = initial
        self.fsm[id] += [self.stateObject]
        self.current = id
        return self
    def addTransition(self, event, action, target):
        if event in self.stateObject['transitions']: raise FsmlDeterminismException;
        self.stateObject['transitions'][event] += \
          [(action, self.current if target is None else target)]
        return self
```

When comparing the Python implementation with the earlier Java implementation, we note that the Python class does not feature members for “observation”; remember the methods getInitial and makeTransition in Illustration 2.9. This is a matter of choice; we assume here that the programmer can simply access the dictionary-based representation of FSMs in the case of Python.

Note: We summarize some important workflows in this book by means of “recipes” such as the one below. The frontmatter of the book features a list of recipes.

Recipe 2.1 (Development of a fluent API).

Samples *Pick some sample DSL "programs" and represent them as expressions in the host language. Strive for fluency by adopting techniques such as method chaining.*

API *Extract the fluent API from the samples. You may represent the API literally as an interface or capture the API by starting the implementation of an object model with empty method bodies.*

Implementation *Identify suitable representation types for the DSL "programs" (e.g., objects with suitable attributes or data structures such as maps). Implement the fluent API in terms of the representation types.*

2.2.3 Interpretation

An obvious aspect of implementing a DSL for FSMs is the *simulation* of FSMs in the sense of processing some input (a sequence of events) to experience the resulting state transitions and to derive the corresponding output (a sequence of actions). At *Acme*, engineers appreciated the possibility of simulation because it would allow them to "play" with the FSMs and to document and verify traces of expected system behavior without yet implementing the FSMs proper on the target platform.

FSM simulation is an instance of what is generally referred to as *interpretation*. An interpreter processes a "program" (i.e., an FSM in the running example), it takes possibly additional input (namely a sequence of events in the running example), and it returns an output (namely a sequence of actions in the running example). We may also use streams to enable "interactive" as opposed to "batch-oriented" simulation.

Let us capture an expected "run" of the turnstile FSM as a (JUnit) testcase.

Illustration 2.11 (Test case for simulation of turnstile execution)

Java source code org/softlang/fsml/tests/FluentTest.java

```
public class FluentTest {

  private static final String[] input =
    {"ticket", "ticket", "pass", "pass", "ticket", "mute", "release"};
  private static final String[] output =
    {"collect", "eject", "alarm", "eject"};

  @Test
  public void runSample() {
    assertArrayEquals(output, run(Sample.turnstile, input));
  }
}
```

In this test case, we invoke a run method with a sequence of events as input (i.e., as a method argument) and with a sequence of actions as output (i.e., as the method result). Both actual input and expected output are set up as string arrays accordingly. The run method (i.e., the FSML interpreter) can be implemented in Java as follows.

Illustration 2.12 (An interpreter for FSML hosted by Java)

Java source code org/softlang/fsml/fluent/FsmlInterpreter.java

```
public class FsmlInterpreter {
  public static String[] run(Fsm fsm, String[] input) {
    ArrayList<String> output = new ArrayList<>();
    String state = fsm.getInitial();
    for (String event : input) {
      ActionStatePair pair = fsm.makeTransition(state, event);
      if (pair.action != null) output.add(pair.action);
      state = pair.state;
    }
    return output.toArray(new String[output.size()]);
  }
}
```

That is, the semantics of an FSM is essentially modeled by the API members getInitial and makeTransition so that it just remains to loop over the input and accumulate the output.

Let us implement an interpreter in Python.

Illustration 2.13 (An interpreter for FSML hosted by Python)

Python module FsmlInterpreter

```
def run(fsm, input):
    # Determine initial state
    for id, [decl] in fsm.iteritems():
        if decl["initial"]:
            current = decl
            break
    # Consume input; produce output
    output = []
    while input:
        event = input.pop(0)
        if event not in current["transitions"]: raise FsmlInfeasibleEventException
        else:
            [(action, target)] = current["transitions"][event]
            if action is not None: output.append(action)
            if target not in fsm: raise FsmlResolutionException
            [current] = fsm[target]
    return output
```

In this implementation, the underlying data structure is accessed directly; this also entails an extra loop to identify the initial state.

The present section is summarized by means of a recipe.

Recipe 2.2 (Development of an interpreter).

Program representation *Set up representation types for the programs to be interpreted. For instance, you may rely on the representation types used by a more or less fluent API (Recipe 2.1).*

Arguments *Identify types of interpretation arguments. In the case of FSML, the interpreter takes a sequence of events, i.e., strings.*

Results *Identify types of interpretation results. In the case of FSML, the interpreter returns a sequence of actions, i.e., strings. The interpreter could also expose intermediate states encountered during the transitions – even though this was not demonstrated earlier.*

Test cases *Set up test cases for the interpreter. A positive test case consists of a program to be interpreted, additional arguments, and the expected result(s). A negative test case does not provide an expected result; instead it is marked with the expectation that interpretation terminates abnormally.*

Case discrimination *Implement interpretation as case discrimination on the syntactic constructs. The interpretation of compound constructs commences recursively or by list processing.*

Testing *Test the interpreter in terms of the test cases.*

We will refine the interpreter recipe in Chapter 5.

Exercise 2.3 (Irregular interpreter completion for FSML) [Basic level]
Implement a test case which illustrates irregular completion. Hint: Design an event sequence such that simulation ends up in a state where a given event cannot be handled.

2.2.4 Well-Formedness

An FSM should meet certain well-formedness constraints to “make sense”, i.e., so that we can expect interpretation of the FSM to be feasible. For instance, each target state mentioned in a transition of an FSM should also be declared in the FSM. Clearly, it is important that language users at *Acme* have a good understanding of these constraints so that they use the language correctly. New *Acme* employees attend an FSML seminar, where they are trained according to the principle “language by example”, i.e., understanding the language by means of complementary, illustrative examples. This includes both well-formed (useful) examples and simple illustrations of constraint violation.

Here is a list of some conceivable constraints; we assign names to the constraints for later reference:

- *distinctStateIds*: The state ids of the state declarations must be distinct.
- *singleInitialState*: An FSM must have exactly one initial state.
- *deterministicTransitions*: The events must be distinct per state.
- *resolvableTargetStates*: The target state of each transition must be declared.
- *reachableStates*: All states must be reachable from the initial state.

Yet more constraints could be identified. For instance, we could require that all states offer transitions for all possible events; this constraint is not met by the turnstile FSM. Let us demonstrate violation of a constraint with an FSM. In the following code, we use the fluent Python API (Section 2.2.2).

Illustration 2.14 (A violation of the *resolvableTargetStates* constraint)

Python module FsmlResolutionNotOk

```
resolutionNotOk = \
  Fsm() \
    .addState("stateA") \
      .addTransition("eventI", "actionI", "stateB") \
      .addTransition("eventII", "actionII", "stateC") \
    .addState("stateB")
```

Exercise 2.4 (Violation of constraints) [Basic level]
Construct an FSM which violates the distinctStateIds *constraint. Construct another FSM which violates the* reachableStates *constraint.*

FSMs exercising constraint violation can be turned into negative test cases for the implementation of the DSL. In implementing the fluent API (Section 2.2.2), we have already shielded against some problems related to the aforementioned constraints. That is, the addState method throws if the given state id has been added before, thereby addressing the constraint *distinctStateIds*. Also, the addTransition method throws if the given event has already occurred in another transition for the current state, thereby addressing the constraint *deterministicTransitions*. The constraints may be implemented by predicates as shown below.

Illustration 2.15 (Constraint checking for FSMs)

Python module FsmlConstraints

```
def ok(fsm):
    for fun in [
            distinctStateIds,
            singleInitialState,
            deterministicTransitions,
            resolvableTargetStates,
            reachableStates ] : fun(fsm)

def distinctStateIds(fsm):
    for state, decls in fsm.iteritems():
        if not len(decls) == 1: raise FsmlDistinctIdsException()

def singleInitialState(fsm):
    initials = [initial for initial, [decl] in fsm.iteritems() if decl["initial"]]
    if not len(initials) == 1: raise FsmlSingleInitialException()

def deterministicTransitions(fsm):
    for state, [decl] in fsm.iteritems():
        for event, transitions in decl["transitions"].iteritems():
            if not len(transitions) == 1: raise FsmlDeterminismException()

def resolvableTargetStates(fsm):
    for _, [decl] in fsm.iteritems():
        for _, transitions in decl["transitions"].iteritems():
            for (_, target) in transitions:
                if not target in fsm: raise FsmlResolutionException()

def reachableStates(fsm):
    for initial, [decl] in fsm.iteritems():
        if decl["initial"]:
            reachables = set([initial])
            chaseStates(initial, fsm, reachables)
    if not reachables == set(fsm.keys()): raise FsmlReachabilityException()

# Helper for recursive closure of reachable states
def chaseStates(source, fsm, states): ...
```

Arguably, some constraints do not need to be checked if we assume a fluent API implementation as discussed before, because some constraint violations may be caught during construction. However, we do not assume necessarily that all DSL samples are constructed by means of the fluent API. For instance, DSL samples may also be represented in interchange formats, thereby calling for well-formedness checking atop serialization.

We mention in passing that additional constraints apply, when all arguments are considered for interpretation. In the case of FSML, we must require that the events in the input can always be handled in the corresponding transition. This sort of problem is caught by the interpreter.

The present section is summarized by means of a recipe.

Recipe 2.3 (Development of a constraint checker).

Negative test cases *Designate one negative test case for each constraint that should be checked. Ideally, each such test case should violate just one constraint and not several at once.*

Reporting *Choose an approach to "reporting". The result of constraint violation may be communicated either as a Boolean value, as a list of error messages, or by throwing an exception.*

Modularity *Implement each constraint in a separate function, thereby supporting modularity and testing.*

Testing *The constraint violations must be correctly detected for the negative test cases. The positive test cases, for example, those for the interpreter (Recipe 2.2), must pass.*

2.3 External DSL

The developers at *Acme* were happy with the internal DSL implementation, as it helped them to experiment with FSMs in a familiar programming language. However, the programming-language notation implied a communication barrier between developers and other stakeholders, who could not discuss matters in terms of programs or did not want to.

An *Acme* developer with competence in language implementation therefore proposed a concise and machine-checkable domain-specific *textual syntax* for FSMs as exercised below.

Illustration 2.16 (Turnstile FSM in textual syntax)

FSML resource 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;
}
```

In the textual notation, all state declarations group together the transitions with the given state as the source state. The target state of a transition appears to the right of the arrow "−>". If the arrow is missing, this is taken to mean that the target state equals the source state.

2.3.1 Syntax Definition

An *Acme* developer with competence in software language engineering suggested a *grammar*-based syntax definition as follows.

```
fsm : state+ EOF ;
state : 'initial'? 'state' stateid '{' transition* '}' ;
transition : event ('/' action)? ('−>' target=stateid)? ';' ;
stateid : NAME ;
event : NAME ;
action : NAME ;
NAME : ('a'..'z'|'A'..'Z')+ ;
```

A variation of the EBNF [7] notation for context-free grammars [6] is used here. The grammar rules define the syntactic categories ("nonterminals"): state machines (fsm), state declarations (state), transitions (transition), and more basic categories for state ids, events, and actions. Each rule consists of the name of the being defined (on the left), a separator (":"), and the actual definition (on the right) in terms of other grammar symbols. For instance, the rule defining fsm models the fact that an FSM consists of a non-empty sequence of state declarations followed by the EOF (end-of-file) character. The rule defining state models the fact that a state declaration starts with the optional keyword 'initial', followed by the keyword "state", followed by a state id, followed by a sequence of transitions enclosed in braces.

Let us provide a general recipe for authoring a grammar.

Recipe 2.4 (Authoring a grammar).

Samples *Sketch the intended language in terms of a few simple samples (i.e., strings) without trying to design a grammar at the same time. If you have carried out a domain analysis (Section 1.3), then your samples should cover the concepts identified by the analysis.*

Categories *Identify the syntactic categories exercised in your samples (and possibly suggested by your domain analysis), for example, state declarations and transitions in the case of FSML. Assign names to these categories. These names are referred to as* nonterminals*, to be defined by the grammar; they show up on the left-hand sides of grammar rules.*

Alternatives *Identify the alternatives for each category. (Again, a domain analysis may readily provide such details.) FSML is so simple that there is* *only a single alternative per category, but think of different expression forms in a language with arithmetic and comparison expressions. Assign names to these alternatives; these names may be used to label grammar rules.*

Structure *Describe the structure of each alternative in terms of* nonterminals, terminals *(keywords and special characters), sequential composition (juxtaposition), repetition ("*" or "+"), and optionality ("?").*

Validation *Ultimately, check that the language samples comply with the authored grammar, as discussed later (Recipe 2.5).*

2.3.2 Syntax Checking

A grammar can be used directly for implementing a *syntax checker* so that everyone can easily check conformance of given text to the rules of the textual syntax. By implementing such a checker, the *Acme* engineers started a transition from an internal to an *external DSL*. That is, there was a dedicated frontend for FSML to permit the representation of FSMs in a language-specific notation without making any concessions to an existing programming language. The *Acme* developer in charge chose the popular technology ANTLR[5] [11] for implementing the syntax checker. That is, ANTLR includes a parser generator which generates code for syntax checking (or parsing) from a given syntax definition (a grammar). The grammar, which was shown above, can be trivially completed into actual ANTLR input so that most of the code for a syntax checker can be generated by ANTLR, as shown below.

Illustration 2.17 (An ANTLR input for FSML)

ANTLR resource languages/FSML/Java/Fsml.g4

```
1  grammar Fsml;
2  @header {package org.softlang.fsml;}
3  fsm : state+ EOF ;
4  state : 'initial'? 'state' stateid '{' transition* '}' ;
5  transition : event ('/' action)? ('->' target=stateid)? ';' ;
6  stateid : NAME ;
7  event : NAME ;
8  action : NAME ;
9  NAME : ('a'..'z'|'A'..'Z')+ ;
10 WS : [ \t\n\r]+ -> skip ;
```

The earlier grammar appears in lines 3–9. Otherwise, the ANTLR input features the following details.

- The grammar is given a name: Fsml (line 1). This name is used in names of generated Java classes such as FsmlParser and FsmlLexer.

[5] http://www.antlr.org/

- By means of a header pragma, a Java package name is specified: org.softlang.fsml (line 2). The generated Java classes are put into this package.
- A special grammar symbol for white space is declared: WS (line 10). Such white space is to be skipped in the input, as controlled by the skip action.
- Two of the nonterminals use uppercase identifiers: NAME and WS (lines 9–10). This is a hint to ANTLR that these nonterminals model lexical syntax. That is, the input text is first converted into a sequence NAME and WS tokens as well as keywords or special tokens from the other rules, before parsing commences.

The present section is summarized by means of a recipe.

Recipe 2.5 (Development of a syntax checker).

Grammar *It is assumed that you have authored a grammar and samples according to Recipe 2.4.*

Approach *Choose an approach to grammar implementation. In this section, we favored the use of a parser generator (ANTLR). In Chapter 7, we will also discuss programmatic implementation (recursive descent and parser combinators).*

Driver *Develop driver code for applying the implemented grammar to input.*

Testing *Apply the syntax checker to language samples to confirm their conformance to the grammar. One should also author samples with syntax errors to test that the syntax checker catches the errors and communicates them appropriately.*

In the running example, we still need the driver code for applying the ANTLR-based checker to samples, as shown below.

Illustration 2.18 (Driver code for the generated syntax checker (parser))

Java source code org/softlang/fsml/FsmlSyntaxChecker.java

```java
public class FsmlSyntaxChecker {
  public static void main(String[] args) throws IOException {
    FsmlParser parser =
      new FsmlParser(
        new CommonTokenStream(
          new FsmlLexer(
            new ANTLRFileStream(args[0]))));
    parser.fsm();
    System.exit(parser.getNumberOfSyntaxErrors()−Integer.parseInt(args[1]));
  }
}
```

The code is idiosyncratic to ANTLR; it entails the following steps:

- An ANTLRFileStream object is constructed and applied to a filename; this is essentially an input stream to process a text file.

- An FsmlLexer object is wrapped around the stream; this is a lexer (scanner) object as an instance of a class that was generated from the grammar.
- A CommonTokenStream object is wrapped around the lexer; thereby allowing the lexer to communicate with the parser in a standardized manner.
- An FsmlParser object is wrapped around the token stream; this is a parser object as an instance of a class that was generated from the grammar.
- The parser is invoked; in fact, the nonterminal (the method) fsm is selected. As a side effect, a parse tree (CST) is associated with the parser object and parse errors, if any, can be retrieved from the same object.
- Finally, there is an assertion to check for parse errors.

The driver code shown is used by a test suite. We have set up the main method in such a way that we can check positive and negative examples through a command-line interface. That is, two arguments are expected: the name of the input file and the expected number of syntax errors. The main method exits with “0”, if the actual number of syntax errors equals the expected number, otherwise it exits with a nonzero code. Let us provide a sample for which syntax checking should fail.

Illustration 2.19 (A syntactically incorrect FSML sample)

FSML resource languages/FSML/tests/syntaxError.fsml

```
initial state stateA {
```

The ANTLR-based parser should report a syntax error as follows:

```
..line 2:0 missing '}' at '<EOF>'
```

For the sake of completeness, let us describe the build process of the ANTLR- and Java-based syntax checker, as it combines code generation and compilation. We may capture the involved steps by means of a Makefile[6], as shown below.

Illustration 2.20 (Makefile for the FSML syntax checker)

Makefile resource languages/FSML/Java/Makefile

```
cp = −cp .:../../../lib/Java/antlr−4.5.3−complete.jar
antlr = java ${cp} org.antlr.v4.Tool −o org/softlang/fsml
fsmlSyntaxChecker = java ${cp} org.softlang.fsml.FsmlSyntaxChecker

all:
  make generate
  make compile
  make test

generate:
  ${antlr} Fsml.g4
```

[6] http://en.wikipedia.org/wiki/Makefile

```
12
13 compile:
14     javac ${cp} org/softlang/fsml/*.java
15
16 test:
17     ${fsmlSyntaxChecker} ../sample.fsml 0
18     ${fsmlSyntaxChecker} ../tests/syntaxError.fsml 1
```

That is:

- Java's classpath is adjusted to incorporate the ANTLR tool and runtime (line 1).
- The invocation of the ANTLR tool for parser generation boils down to running the main method of the Java class org.antlr.v4.Tool from the ANTLR jar with some option ("-o") for the output directory (line 2 for the command line and line 11 for the actual application).
- The invocation of the syntax checker for FSML boils down to running the main method of the Java class org.softlang.fsml.FsmlSyntaxChecker (line 3 for the command line and lines 17–18 for the actual application). Each invocation involves the input file to be checked and the number of expected syntax errors.

By performing syntax checking at *Acme*, some level of quality assurance regarding the DSL for FSMs was supported. Language users could make sure that their samples conformed to the intended syntax.

2.3.3 Parsing

Now let us suppose that we want to process the textual input on the basis of its grammar-based structure. Thus, we need to make a transition from syntax checking (or "acceptance") to parsing. Classically, the output of parsing is a parse tree or concrete syntax tree (CST), the structure of which is aligned with the underlying grammar. A parser may also perform abstraction to eliminate details that are not relevant for assigning semantics to the input. In this case, the output of parsing is an abstract syntax tree (AST) or an abstract syntax graph (ASG), if the parser additionally performs resolution to discover references in the otherwise tree-based syntactical structure. The output of parsing is also referred to as a "model" in the context of model-driven engineering (MDE). The term "text-to-model" (transformation) may be used instead of "parsing" in the MDE context.

At *Acme*, it was decided that the parser should construct ASTs such that an existing object model for FSMs was used for the representation of ASTs. In this manner, one would be able also to apply well-formedness checking and interpretation (indirectly) to FSMs that are represented as text. This gives rise to the notion of "text-to-objects". The grammar of the ANTLR-based syntax checker was reused. ANTLR support for so-called parse-tree listeners was leveraged to attach functionality to the grammar for the construction of suitable objects.

An ANTLR listener is a collection of programmer-definable handler methods that are invoked by the parsing process at well-defined points. There are, basically, methods for entering and exiting parse-tree nodes for any nonterminal of the grammar. In fact, the methods are invoked during a generic walk over a parse tree that ANTLR constructs during its generic parsing process. Given a grammar, ANTLR generates a suitable listener class (FsmlBaseListener in the present example) with empty handler methods. A programmer may extend the base listener by implementing handler methods that perform object construction. Let us present a listener which facilitates parsing FSMs into objects according to the baseline object model for FSML (Section 2.2.1). The corresponding Java code follows.

Illustration 2.21 (A parse-tree listener for text-to-objects)

Java source code org/softlang/fsml/FsmlToObjects.java

```
public class FsmlToObjects extends FsmlBaseListener {
  private Fsm fsm;
  private State current;
  public Fsm getFsm() { return fsm; }
  @Override public void enterFsm(FsmlParser.FsmContext ctx) {
    fsm = new Fsm();
  }
  @Override public void enterState(FsmlParser.StateContext ctx) {
    current = new State();
    current.setStateid(ctx.stateid().getText());
    fsm.getStates().add(current);
  }
  @Override public void enterTransition(FsmlParser.TransitionContext ctx) {
    Transition t = new Transition();
    fsm.getTransitions().add(t);
    t.setSource(current.getStateid());
    t.setEvent(ctx.event().getText());
    if (ctx.action() != null) t.setAction(ctx.action().getText());
    t.setTarget(ctx.target != null ? ctx.target.getText() : current.getStateid());
  }
}
```

Thus, the listener extends FsmlBaseListener and it overrides enterFsm, enterState, and enterTransition – these are the events of entering parse-tree nodes rooted in the rules for the nonterminals fsm, state, and transition. The methods construct an FSM object, which is stored in the attribute fsm of the listener.

We also need driver code to compose syntax checking, parse-tree construction (done transparently by the ANTLR runtime), and parse-tree walking with the listener at hand, as shown below.

Illustration 2.22 (Parsing with an ANTLR listener)

Java source code org/softlang/fsml/tests/FsmlToObjectsTest.java

```java
public Fsm textToObjects(String filename) throws IOException {
  FsmlParser parser = new FsmlParser(
      new CommonTokenStream(
          new FsmlLexer(
              new ANTLRFileStream(filename))));
  ParseTree tree = parser.fsm();
  assertEquals(0, parser.getNumberOfSyntaxErrors());
  FsmlToObjects listener = new FsmlToObjects();
  ParseTreeWalker walker = new ParseTreeWalker();
  walker.walk(listener, tree);
  return listener.getFsm();
}
```

This process consists of these phases:

- We construct an FsmlParser object and thus also objects for a file stream, a lexer, and a token stream (lines 2–5). We use the same ANTLR grammar and the same generated code as for the syntax checker.
- We invoke the parser (line 6). During parsing the parse tree is constructed and is returned as the result of the method call parser.fsm().
- We check that parsing has completed without errors (line 7), as it would not be sound to access the parse tree otherwise.
- We construct an FsmlToObjects object for listening (line 8), as explained earlier.
- We construct a ParseTreeWalker object (line 9) and we invoke the walker's walk method while passing the listener and the parse tree as arguments (line 10).
- Ultimately, we can extract the constructed AST from the listener object (line 11).

Let us summarize the development steps for obtaining a parser (i.e., a text-to-model or text-to-objects transformation); the recipe given below mentions ANTLR and its listener-based approach while it characterizes the underlying steps also more generally.

Recipe 2.6 (Development of a parser).

Syntax checker *Develop a syntax checker for the language according to Recipe 2.5.*

Representation *Design a representation for parse trees, unless a suitable representation is readily provided by the underlying technology such as a parser generator. The representation may be defined, for example, in terms of an object model, by means of JSON, or by other means of abstract syntax implementation (Recipe 4.1).*

Parse trees *Implement functionality for the construction of parse trees, unless a suitable representation is readily constructed by the underlying tech-*

nology. For instance, in the case where ANTLR is used, you may implement a listener for mapping generic ANTLR-specific parse trees to a designated object model.

Driver *Generalize the driver code of the underlying syntax checker to perform parsing, i.e., mapping text to parse trees.*

Testing *Generalize the test suite of the underlying syntax checker to perform parsing, including the validation of the returned parse trees by comparison with baselines.*

Exercise 2.5 (Validation of text-to-objects) [Intermediate level]
How would you validate that the parser obtained according to Recipe 2.6 constructs reasonable ASTs? To this end, assume that there are a large number of valid textual inputs available. You need to find a scalable approach that takes into account all these inputs.

2.4 DSL Services

Arguably, we have reached the "minimum" of a language implementation: representation (internal style and grammar-based textual syntax), parsing, interpretation, and well-formedness checking. In practice, a DSL is likely to call for yet other language-based components or "services". For the running example, we are going to discuss briefly an interchange format for serializing FSMs, a visual syntax for FSML, and (C) code generation to represent FSMs directly as executable code. Examples of yet other language services, which, however, are not discussed here, include these: a refactoring tool for FSMs (e.g., for renaming state ids), a generator tool for FSMs that could be used to test language services, a language-specific editor, other IDE services, and a verification tool that could be used to prove equivalence or subsumption for FSMs.

2.4.1 Interchange Format

At *Acme*, the developers wanted to implement language-based components in different programming languages while permitting integration of the services on the basis of an interchange format for serialization. For instance, it should be possible to use the output of a Java-based parser in a Python-based well-formedness checker. An interchange format would also make it possible to distribute the language implementation, for example, in a web application. The *Acme* developers agreed on a JSON[7]-based representation as follows.

[7] `http://json.org/`

Illustration 2.23 (A JSON-based model of the turnstile FSM)

JSON resource languages/FSML/Python/tests/baselines/sample.json

```
{"exception": [{
     "initial": false,
     "transitions": {
          "release": [[null, "locked"]],
          "ticket": [["eject", "exception"]],
    "mute": [[null, "exception"]],
    "pass": [[null, "exception"]]}}],
 "locked": [{
      "initial": true,
      "transitions": {
        "ticket": [["collect", "unlocked"]],
        "pass": [["alarm", "exception"]]}}],
 "unlocked": [{
      "initial": false,
      "transitions": {
        "ticket": [["eject", "unlocked"]],
        "pass": [[null, "locked"]]}}]}
```

JSON is suitable for language-agnostic representation of (nested) dictionary-like data with support for lists and some data types. JSON-based serialization is supported for most, if not all, popular programming languages. In the JSON-based representation of an FSM, as shown above, an FSM is a nested dictionary with the state ids as keys at the top, with keys "initial" and "transitions" per state, and with the events as keys per transition. For each event, a pair consisting of an action ("null" when missing) and a target state is maintained. In fact, each event is mapped to a list of action-state pairs; see the following exercise.

Exercise 2.6 (Lists of action-state pairs) [Basic level]
What "expressiveness" is gained by mapping events to lists of action-state pairs? Hint: Think of the separation of parsing and well-formedness checking.

The rules underlying the JSON format may be understood as defining the abstract syntax of FSML. The engineers at *Acme* did not bother to define the format explicitly by means of a schema, but this would be possible; see the following exercise.

Exercise 2.7 (A JSON schema for FSML) [Intermediate level]
Using the example model in Illustration 2.23 and the informal explanations of the format, define a schema in JSON Schema[8] for FSML. Perform schema-based validation on the example.

[8] http://json-schema.org/

Exercise 2.8 (A JSON exporter for Java objects) [Intermediate level]
Implement an object-to-JSON mapping in Java. Start from the baseline object model for FSML (Section 2.2.1). Check that the mapping results in the expected JSON output for the turnstile example.

Exercise 2.9 (Integrating Java and Python components) [Intermediate level]
Use the Java-based parser of Illustration 2.22 to parse text into objects. Use the Java-based JSON exporter of Exercise 2.8 to serialize objects as JSON. It turns out that the JSON format, when deserialized into Python with the "standard" load function, fits exactly the representation type of the fluent API implementation in Illustration 2.7. Validate the suitability of the Python objects, thus obtained, by applying the Python-based components for interpretation and well-formedness checking, as discussed earlier.

There are various alternatives to a JSON-based interchange format. Other possible options include XML[9] and ASN.1.[10]

2.4.2 Code Generation

In the recent past, *Acme* engineers discovered that they could use FSMs for generating part of the ultimate implementation. In fact, as FSMs are used at *Acme* for many different purposes and on many different devices and platforms, several code generators were developed over time. Prior to using *code generation*, FSMs were manually implemented in a more or less idiomatic manner.

In principle, one could "execute" FSMs on the target platform by means of some form of (interactive) interpretation. However, code generation complements interpretation in several ways:

Efficiency The generated code may potentially be more efficient than interpretation, just in the same way as compiled code typically runs faster than interpreted code. The execution of the compiled code may also require less runtime resources than interpretation. In particular, the interpreter itself, including its data structures would not be needed for running the generated code.

Pluggability Developers may need to plug actual functionality into FSM execution. For instance, events and actions are merely "symbols" in FSML, but actual functionality needs to be executed on the target platform so that FSM execution interacts with sensors and actors. Such pluggability is also feasible with interpretation, but perhaps even more straightforward with generated code.

[9] `http://www.w3.org/XML/`

[10] `http://en.wikipedia.org/wiki/Abstract_Syntax_Notation_One`

Customizability The actual implementation of behavior, as specified by the FSM, may need customization in some way. For instance, specific conditions may need to be added on transitions and extra housekeeping may need to be arranged to this end. By representing FSMs within a programming language, the programmers may customize functionality in a familiar manner.

Let us develop a simple code generator. Let us assume here that neither Python nor Java is supported on the target platform, which may be a lower-level platform for embedded systems, but there exists a *C* compiler emitting code for the target platform. Thus, our code generator must generate *target code* in the C language (rather than in Java or Python). Before looking at the implementation of the generator, let us agree on a baseline for the generated code, as shown below.

Illustration 2.24 (Generated code for the turnstile FSM)

C resource languages/FSML/Python/generated/Turnstile.c

```c
enum State { EXCEPTION, LOCKED, UNDEFINED, UNLOCKED };
enum State initial = LOCKED;
enum Event { RELEASE, TICKET, MUTE, PASS };
void collect() { }
void alarm() { }
void eject() { }
enum State next(enum State s, enum Event e) {
 switch(s) {
  case EXCEPTION:
   switch(e) {
    case RELEASE: return LOCKED;
    case TICKET: eject(); return EXCEPTION;
    case PASS: return EXCEPTION;
    case MUTE: return EXCEPTION;
    default: return UNDEFINED;
   }
  case LOCKED:
   switch(e) {
    case TICKET: collect(); return UNLOCKED;
    case PASS: alarm(); return EXCEPTION;
    default: return UNDEFINED;
   }
  case UNLOCKED:
   switch(e) {
    case TICKET: eject(); return UNLOCKED;
    case PASS: return LOCKED;
    default: return UNDEFINED;
   }
  default: return UNDEFINED;
 }
}
```

The C code contains these elements:

- An enumeration type for the state ids (line 1).
- A declaration for the initial state (line 2).
- An enumeration type for the events (line 3).
- Functions for the actions with empty bodies (lines 4–6).
- A function next (lines 7–31) which takes the current state s and an event e, performs the corresponding action, if any, and returns the new state. This function is defined by a nested switch-statement that dispatches on s and e.

It is up to the developer of the embedded system to wire up the generated code to the functionality for accessing sensors (to observe events) and actors (to perform actions).

Exercise 2.10 (Representation options) [Intermediate level]
There are several options for code-level representations of FSM transitions: (i) a cascaded switch-statement, as in Illustration 2.24; (ii) a data structure using appropriate data types for collections, as used in the Java-based implementation of the fluent API in Illustration 2.9; and (iii) an OO approach with an abstract base type for states and one concrete subtype per state so that a polymorphic method for state transitions takes the current event and selects an action as well as the target state. What are the tradeoffs of these options, when using the following dimensions for comparison: runtime efficiency, runtime adaptiveness, type safety for generated code, and simplicity of the code generator? (You may need to actually experiment with code generators for the options.)

Let us leverage *template processing* to generate the required C code. The pattern of the code to be generated is represented by a *template*. Template processing boils down to instantiation of templates, i.e., parameterized text, in a program.

One *Acme* developer decided to exercise template processing in Python and to leverage the template engine *Jinja2*[11]. The template is shown below.

Illustration 2.25 (Jinja2-based template for C code for FSM)

Jinja2/C resource languages/FSML/Python/templates/Fsm.jinja2

```
enum State { {{states|join(', ')|upper()}} };
enum State initial = {{initial|upper}};
enum Event { {{events|join(', ')|upper()}} };
{% for a in actions %}void {{a}}() { }
{% endfor %}
enum State next(enum State s, enum Event e) {
 switch(s) {
{% for (s, ts) in transitions %}
  case {{s|upper()}}:
```

[11] http://jinja.pocoo.org/

```
10    switch(e) {
11 {% for (e, a, t) in ts %}
12     case {{e|upper()}}: {% if a %}{{a}}(); {% endif %}return {{t|upper()}};
13 {% endfor %}
14     default: return UNDEFINED;
15    }
16 {% endfor %}
17   default: return UNDEFINED;
18  }
19 }
```

It is best to compare the template with an instance; see again Illustration 2.24. The following concepts are used:

- For as long as the template does not involve templating-specific constructs, the template's text is literally copied to the output. For instance, the header of the method next (line 6) is directly copied from the template to the output.
- A template is parameterized by (Python) data structures that the template may refer to. For instance, there are Jinja2-level for-loops (lines 8 and 11) in the template which loop over parameters such as actions and transitions to generate similar code for all elements of these collection-typed parameters.
- The text content of a parameter, say x, can be inlined by using the notation "$\{\{x\}\}$ where x" is a parameter. Parameters either are directly passed to the template or are extracted from other parameters, for example, within for-loops.
- Some parameters are processed by so-called filters; see the occurrences of upper and join. In this manner, the raw text of parameters is manipulated. That is, join composes a list of strings by interspersing another string (here, a comma); upper turns a string into uppercase.

There is more expressiveness for template processing, but we omit a detailed discussion here. The only missing part of the code generator is the functionality for template instantiation as shown below.

Illustration 2.26 (Template instantiation)

Python module FsmlCGenerator

```
def generateC(fsm):
    # Initialize data structures
    states = set()
    states.add("UNDEFINED")
    events = set()
    actions = set()
    transitions = list()
    # Aggregate data structures
    for source, [statedecl] in fsm.iteritems():
        ts = list()
        transitions.append((source, ts))
        states.add(source)
        if statedecl["initial"]:
            initial = source
        for event, [(action, target)] in statedecl["transitions"].iteritems():
            events.add(event)
            if action is not None: actions.add(action)
            ts.append((event, action, target))
    # Look up template
    env = Environment(loader=FileSystemLoader('templates'), trim_blocks=True)
    fsmTemplate = env.get_template('Fsm.jinja2')
    # Instantiate template
    return fsmTemplate.render(\
            states = states,\
            initial = initial,\
            events = events,\
            actions = actions,\
            transitions = transitions)
```

Thus, the template parameters states, events, actions, and transitions are trivially synthesized from the Python objects. Other than that, the code for template instantiation loads the template and renders it.

Another *Acme* developer decided to exercise template processing in Java and to leverage the template engine *StringTemplate*[12] [10]. StringTemplate encourages the use of template *groups*, that is, templates that invoke each other, as shown for FSML below.

Illustration 2.27 (StringTemplate-based templates for C code for FSM)

StringTemplate/C resource languages/FSML/Java/templates/Fsm.stg

```
main(states, initial, events, actions, tgroups) ::= <<
enum State { <states; format="upper", separator=", "> };
enum State initial = <initial; format="upper">;
enum Event { <events; format="upper", separator=", "> };
<actions:action(); format="lower", separator="\n">
enum State next(enum State s, enum Event e) {
    switch(s) {
<tgroups:tgroup(); separator="\n">
        default: return UNDEFINED;
    }
}>>

action(a) ::= "void <a>() { }"

tgroup(g) ::= <<
        case <g.stateid; format="upper">:
            switch(e) {
                <g.ts:transition(); separator="\n">
                default: return UNDEFINED;
            }>>

transition(t) ::= <%
```

[12] http://www.stringtemplate.org/

```
23 case <t.event; format="upper">:
24 <if(t.action)><t.action; format="lower">(); <endif>
25 return <t.target; format="upper">;%>
```

Let us explain the StringTemplate notation.

- We use a definition form $templ(p)$ ::= "…" to define named templates with parameters (such as p) that can invoke each other, just like possibly recursive functions. There is a *main* template (lines 1–11) to start template processing with. There is an *action* template (line 13) for the C code for each action function. There is a *tgroup* template (lines 15–20) for the transitions grouped by source state. There is also a *transition* template (lines 22–25) for the code for a single transition.
- We use $< \%.\dots\% >$ instead of "…" to define multi-line instead of single-line templates.
- We use $<<$ ... $>>$ instead to define multi-line templates. Compared to $< \%.\dots\% >$, indentation and line breaks are transported from the template to the output.
- We use the form $< p >$ to refer to a parameter p, i.e., to inline it as text. We use the form $p : templ()$ to invoke a template *templ* and to pass the parameter p. We use the form $< p.x >$ to refer to the property x of p.
- There are also *format* and *separator* controls that are similar to the filters of Jinja2. There is also expressiveness (if … endif) for conditional parts, just as in the case of Jinja2.

We omit the Java code for template instantiation; it is very similar to the Python code discussed earlier.

Exercise 2.11 (A more advanced code generator) [Basic level]
Revise the code generator so that the generated methods for the FSM actions get access to the initiating event, the source state, and the target state. The idea here is that the plugged code for actions may use these additional parameters for richer behavior. This context should be passed by regular arguments to the methods for the actions.

Exercise 2.12 (An object model for C code) [Intermediate level]
Set up an object model for the subset of C needed in the FSML example. Implement a template-processing component for rendering C code. Implement a mapping between the object models of FSML and C. In this manner, you could implement the code generator in an alternative manner.

The present section is summarized by means of a recipe.

Recipe 2.7 (Development of a code generator).

Test cases *Develop code samples to be generated and complete them into test cases by also listing the corresponding inputs (programs) from which the code samples are to be generated. Strive for simplicity – certainly in the beginning so that code generation is more easily set up. Test that the samples compile and run on the target platform.*

Templates *Parameterize the code samples to obtain templates with appropriate parameters, loops, etc.*

Data structure *Design the data structure for the template parameters. The basic assumption is that some existing representation types (e.g., an object model) may be appropriate.*

Instantiation *Implement the template instantiation functionality such that the data structure for the template parameters is synthesized, templates are loaded, template parameters are assigned, and rendering is done.*

Testing *Test the code generator to return the expected code according to the test cases. Some fine tuning of the templates or the expected output may be required, for example, if spaces, line breaks, and indentation are taken into account.*

2.4.3 Visualization

While the *Acme* engineers agreed on using textual notation for maintaining FSMs throughout the development cycle, some *Acme* employees insisted that a visual notation would still be needed. In particular, several architects made the point that the visual notation was more suitable for meetings with customers. Accordingly, it was decided to provide visualization functionality such that FSMs could be rendered according to a visual syntax; see Fig. 2.2. The notation is inspired by the whiteboard notation of Section 2.1 (Fig. 2.1). It was also decided that no graphical editor was required, because just rendering FSMs would be sufficient. We mention in passing that some competitors of *Acme* use graphical editors for FSMs, as visual syntax is favored in those companies.

The *Acme* engineer in charge decided that the visualization should be based on the popular technology Graphviz.[13] Graphviz processes input which conforms to the so-called DOT language, with language elements for describing graphs in terms of nodes and edges, as well as various attributes that control details of appearance, as shown below.

[13] http://www.graphviz.org/

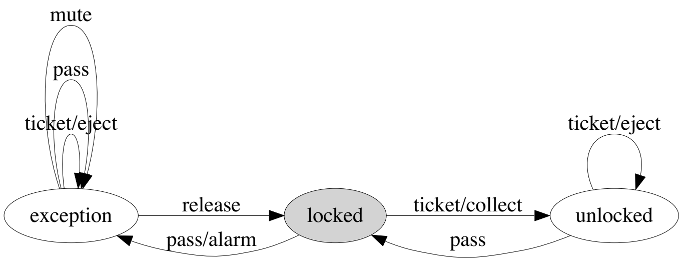

Fig. 2.2 A turnstile FSM in visual notation.

Illustration 2.28 (DOT representation of turnstile FSM)

DOT resource languages/FSML/Python/dot/sample.dot

```
digraph {
  graph [nodesep=0.5,
    rankdir=LR,
    title="Sample FSM"
  ];
  exception [shape=ellipse];
  exception −> exception [label="ticket/eject"];
  exception −> exception [label=pass];
  exception −> exception [label=mute];
  locked [shape=ellipse,
    style=filled];
  exception −> locked [label=release];
  locked −> exception [label="pass/alarm"];
  unlocked [shape=ellipse];
  locked −> unlocked [label="ticket/collect"];
  unlocked −> locked [label=pass];
  unlocked −> unlocked [label="ticket/eject"];
}
```

As FSMs are essentially also just node- and edge-labeled graphs, the visualization functionality should be straightforward. Nevertheless, this functionality is interesting in that it allows us to revisit some of the DSL concepts discussed earlier.

Different techniques may be employed for generating the Graphviz input for FSMs. An obvious option is to leverage templates (Section 2.4.2) such that the DOT graph is obtained by instantiating a template that represents the relevant DOT patterns. Another option is to leverage a DOT API, in fact, an implementation of DOT as an internal DSL such that the DOT graph is constructed by a sequence of API calls. Some pros ("+") and cons ("-") may be identified:

- Use template processing for DOT-graph construction:
  + Relevant DOT constructs are clearly depicted in the template.
  – DOT's syntax may be violated by the template or the instantiation.
- Use a DOT API instead:
  + The resulting DOT graphs are syntactically correct by construction.
  – One needs to understand a specific API.

The following code illustrates the API option. The functionality is straightforward in that it simply traverses the FSM representation and adds nodes and edges to a graph object.

Illustration 2.29 (A visualizer for FSML)

Python module FsmlVisualizer

```
import pygraphviz

def draw(fsm):
    # Create graph
    graph = pygraphviz.AGraph(title="Sample FSM", directed=True, strict=False, rankdir
         ='LR', nodesep=.5)
    # Create nodes
    for fromState, [stateDeclaration] in fsm.iteritems():
        if stateDeclaration["initial"]:
            graph.add_node(n=fromState, shape='ellipse', style='filled')
        else:
            graph.add_node(n=fromState, shape='ellipse')
    # Create edges
    for fromState, [stateDeclaration] in fsm.iteritems():
        for symbol, [(action, toState)] in stateDeclaration["transitions"].iteritems():
            graph.add_edge(fromState, toState, label=symbol + ("" if action is None else
                 "/"+action))
    return graph
```

Exercise 2.13 (Template-based visualization) [Basic level]
Reimplement the visualizer in Illustration 2.29 with template processing instead of using an API for DOT graphs.

Summary and Outline

We have developed the domain-specific language FSML for modeling, simulating, and otherwise supporting finite state machines. Several aspects of language design and implementation were motivated by reference to language users and implementers whom we envisaged, as well as possible changes to requirements over time. The implementation leveraged the programming languages Java, Python, and C as well as additional tools, namely the parser generator ANTLR, the template processors Jinja2 and StringTemplate, and Graphviz with its DOT language.

Clearly, FSML, or any other DSL for that matter, could be implemented in many other ways, within different technological spaces, leveraging different kinds of metaprogramming systems. The online resources of the book come with several alternative implementations. FSML is going to serve as a running example for the remainder of the book.

FSML's language design could be modified and enhanced in many ways. For instance, FSML is clearly related to statecharts in the widely adopted modeling language UML. The statecharts of UML are much more expressive. There is also existing support for statecharts, for example, in terms of code generators in the MDE context. This may suggest a critical discussion to identify possibly additional expressiveness that would also be useful at *Acme*. Also, perhaps, existing UML tooling could provide a more standardized replacement for *Acme*'s proprietary DSL.

In the remaining chapters of this book, we will study the foundations and engineering of syntax, semantics, types, and metaprogramming for software languages. FSML will show up as an example time and again, but we will also discuss other software languages.

References

1. Erdweg, S.: Extensible languages for flexible and principled domain abstraction. Ph.D. thesis, Philipps-Universität Marburg (2013)
2. Erdweg, S., van der Storm, T., Völter, M., Boersma, M., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G.D.P., Molina, P.J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V.A., Visser, E., van der Vlist, K., Wachsmuth, G., van der Woning, J.: The state of the art in language workbenches – conclusions from the language workbench challenge. In: Proc. SLE, *LNCS*, vol. 8225, pp. 197–217. Springer (2013)
3. Erdweg, S., van der Storm, T., Völter, M., Tratt, L., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G.D.P., Molina, P.J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V.A., Visser, E., van der Vlist, K., Wachsmuth, G., van der Woning, J.: Evaluating and comparing language workbenches: Existing results and benchmarks for the future. Comput. Lang. Syst. Struct. **44**, 24–47 (2015)
4. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison Wesley (1999)
5. Fowler, M.: Domain-Specific Languages. Addison-Wesley (2010)
6. Hopcroft, J., Motwani, R., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Pearson (2013). 3rd edition
7. ISO/IEC: ISO/IEC 14977:1996(E). Information Technology. Syntactic Metalanguage. Extended BNF. (1996). Available at `http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf`
8. Kats, L.C.L., Visser, E.: The Spoofax language workbench. In: Companion SPLASH/OOPSLA, pp. 237–238. ACM (2010)
9. Kats, L.C.L., Visser, E.: The Spoofax language workbench: rules for declarative specification of languages and IDEs. In: Proc. OOPSLA, pp. 444–463. ACM (2010)
10. Parr, T.: A functional language for generating structured text (2006). Draft. `http://www.stringtemplate.org/articles.html`
11. Parr, T.: The Definitive ANTLR 4 Reference. Pragmatic Bookshelf (2013). 2nd edition

12. Renggli, L.: Dynamic language embedding with homogeneous tool support. Ph.D. thesis, Universität Bern (2010)
13. Visser, E., Wachsmuth, G., Tolmach, A.P., Neron, P., Vergu, V.A., Passalaqua, A., Konat, G.: A language designer's workbench: A one-stop-shop for implementation and verification of language designs. In: Proc. SPLASH, Onward!, pp. 95–111. ACM (2014)
14. Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L.C.L., Visser, E., Wachsmuth, G.: DSL Engineering – Designing, Implementing and Using Domain-Specific Languages. dslbook.org (2013)
15. Voelter, M., Ratiu, D., Kolb, B., Schätz, B.: mbeddr: instantiating a language workbench in the embedded software domain. Autom. Softw. Eng. **20**(3), 339–390 (2013)
16. Völter, M., Visser, E.: Language extension and composition with language workbenches. In: Companion SPLASH/OOPSLA, pp. 301–304. ACM (2010)
17. Wachsmuth, G., Konat, G.D.P., Visser, E.: Language design with the Spoofax language workbench. IEEE Softw. **31**(5), 35–43 (2014)