UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

Software Language Engineering Techniques

in Language Workbenches and Metaprogramming Languages

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science
im Studiengang Informatik

vorgelegt von
Simon Schauß

Erstgutachter: Prof. Dr. Ralf Lämmel
Institut für Softwaretechnik

Zweitgutachter: M.Sc. Marcel Heinz
Institut für Softwaretechnik

Koblenz, Oktober 2015

Erklärung

Hiermit bestätige ich, dass die vorliegende Arbeit von mir selbständig verfasst wurde und ich keine anderen als die angegebenen Hilfsmittel — insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen — benutzt habe und die Arbeit von mir vorher nicht in einem anderen Prüfungsverfahren eingereicht wurde. Die eingereichte schriftliche Fassung entspricht der auf dem elektronischen Speichermedium (CD-Rom).

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☑	☐
Der Veröffentlichung dieser Arbeit im Internet stimmt ich zu.	☑	☐

....

(Ort, Datum) (Unterschrift)

Zusammenfassung

Implementieren von Softwaresprachen und den dazugehörigen Werkzeugen für diese Sprachen impliziert einen erheblichen Aufwand für Programmierer.

Einerseits zeigt diese Arbeit wie domänenspezifische Sprachen in existierende universell einsetzbare Programmiersprachen mit Hilfe von Metaprogrammierung eingebettet werden können. Andererseits wird veranschaulicht, wie sich der sprachorientierte Programmieransatz von Language Workbenches einsetzen lässt um Verifikationsinfrarstrukturen, Quellcodegeneratoren und integrierte Entwicklungsumgebungen zu entwicklen.

Weiter werden die Features identifiziert, welche von diesen Programmiersprachen und Language Workbenches benutzt werden, um dazugehörige Feature Models zu konstruieren. Diese Modelle, welche sich nicht allein auf die Implementierungen von domänenspezischen Sprachen beschränken, geben einen generellen Überblick über die verwendeten Implementierungstechniken.

Abstract

Implementing software languages and building appropriate tools for these languages implicate great effort for the programmers.

This thesis shows, on the one hand how to embed a domain-specific language into existing general-purpose programming languages utilizing metaprogramming facilities. On the other hand it illustrates how to use the language-oriented programming approach of language workbenches to build verification infrastructures, source code generators and integrated development environments.

Further, the features used by these programming languages and language workbenches are identified to build corresponding feature models. These models give a general view on the different techniques that are used for, but not limited to, the implementation of domain-specific languages.

Acknowledgment

I would like to express my sincere thanks to my supervisor Prof. Dr. Ralf Lämmel for his support and confidence. Also my special thanks go to my parents, for their support throughout my education.

Contents

1 Introduction **1**
1.1 Research Questions . . . 1
1.2 Contributions . . . 1

2 Related Work **3**

3 Implementation **5**
3.1 MPS . . . 5
3.1.1 Abstract Syntax . . . 5
3.1.2 Concrete Syntax . . . 5
3.1.3 Constraints . . . 6
3.1.4 Code Generation . . . 6
3.1.5 Tests . . . 6
3.2 Rascal . . . 7
3.2.1 Syntax Definition . . . 7
3.2.2 Constraint Checking . . . 8
3.2.3 Code Generation . . . 10
3.2.4 Pretty Printing . . . 11
3.2.5 IDE Construction . . . 12
3.2.6 Tests . . . 13
3.3 Spoofax . . . 14
3.3.1 Syntax Definition . . . 14
3.3.2 Name Binding . . . 15
3.3.3 Constraint Checking . . . 15
3.3.4 Code Generation . . . 16
3.3.5 IDE Construction . . . 17
3.3.6 Tests . . . 17
3.4 Haskell . . . 17
3.4.1 Abstract Syntax Tree . . . 17
3.4.2 Concrete Syntax . . . 18
3.4.3 Parser . . . 18
3.4.4 Quasiquotation . . . 19
3.4.5 Constraint Checking . . . 20
3.4.6 Tests . . . 21
3.5 Racket . . . 22
3.5.1 Concrete Syntax . . . 22
3.5.2 Macros . . . 22
3.5.3 Syntax Object Transformation . . . 23

3.5.4 Constraint Checking 24
3.5.5 Tests 24
3.6 Scala 25
3.6.1 Abstract Syntax 25
3.6.2 Macro 26
3.6.3 Constraint Checking 28
3.6.4 Tests 29

4 Feature Models **31**
4.1 Language Workbenches 31
4.1.1 MPS 31
4.1.2 Rascal 31
4.1.3 Spoofax 32
4.1.4 Combined Feature Model 32
4.2 Metaprogramming Languages 34
4.2.1 Haskell 34
4.2.2 Racket 34
4.2.3 Scala 34
4.2.4 Combined Feature Model 35

5 Conclusion **37**

List of Figures

3.1 Structure for the Fsm Node in MPS 5
3.2 Editor for the Fsm Structure in MPS 6
3.3 Non-typesystem Rule for the Fsm Structure in MPS 6
3.4 Reduction Rule for the Fsm Structure in MPS 6
3.5 Tests in MPS 7

4.1 Feature Model for Haskell 31
4.2 Feature Model for Rascal 32
4.3 Feature Model for Spoofax 32
4.4 Feature Model for Language Workbenches 33
4.5 Feature Model for Haskell 34
4.6 Feature Model for Racket 35
4.7 Feature Model for Scala 35
4.8 Feature Model for Metaprogramming 36

Listings

3.1 Syntax definition in Rascal 7
3.2 Check for single initial state constraint in Rascal 8
3.3 Check for distinct state IDs constraint in Rascal 8
3.4 Check for resolvable state ID constraint in Rascal 9
3.5 Check for deterministic input constraint in Rascal 9
3.6 Check for reachable state constraint in Rascal 10
3.7 Code generation in Rascal 10
3.8 Pretty printing in Rascal 11
3.9 Plugin 12
3.10 Annotations 12
3.11 Outline 13
3.12 Tests in Rascal 13
3.13 Syntax definition in Spoofax 14
3.14 Name Binding in Spoofax 15
3.15 Check for single initial state constraint in Spoofax 16
3.16 Check for reachable state constraint in Spoofax 16
3.17 Code generation in Spoofax 16
3.18 Tests in Spoofax 17
3.19 Abstract syntax tree in Haskell 17
3.20 Concrete syntax tree in Haskell 18
3.21 Parser in Haskell 18
3.22 Quoter in Haskell 20
3.23 Check for single initial state constraint in Haskell 20
3.24 Check for deterministic input constraint in Haskell 21
3.25 Check for reachable state constraint in Haskell 21
3.26 Tests in Haskell 22
3.27 Macros in Racket 22
3.28 Macros in Racket 22
3.29 Syntax Object Transformation in Racket 23
3.30 Check for single initial state constraint in Racket 24
3.31 Test helper in Racket 25
3.32 Test case in Racket 25
3.33 Fsm Abstract Syntax node in Scala 25
3.34 State Abstract Syntax node in Scala 25
3.35 Transition Abstact Syntax Node in Scala 26
3.36 Macro definition in Scala 26
3.37 Implicit Conversion in Scala 26
3.38 Unlifting in Scala 26
3.39 Lifting in Scala 27

3.40 Tree Deconstruction in Scala 28
3.41 Tree Construction in Scala 28
3.42 Check for single initial state constraint in Scala 28
3.43 Tests in Scala 29

Chapter 1

Introduction

Developing domain-specific languages accompanied with support by appropriate integrated development environment involves a great deal of expense. Using language workbenches, to implement such domain-specific languages and tools, reduce the burden on programmers by providing software development tools for language oriented programming.

Instead of language workbenches, metaprogramming languages can be utilized to decrease the overhead by embedding domain-specific languages into existing general-purpose programming languages and take advantage of existing software development tools.

This work presents the techniques used by language workbenches and general-purpose programming languages with metaprogramming facilities to develop domain-specific languages.

1.1 Research Questions

The thesis research questions are about, which features are provided by language workbenches and metaprogramming languages to implement a domain-specific language. This leads to the problem how of finding features actually used by implementations of domain-specific languages.

Thus the research questions are:

- How is a domain-specific language implemented with the help of
  - language workbenches?
  - metaprogramming languages?
- What features are utilized to accomplish the implementation by
  - language workbenches?
  - metaprogramming languages?

1.2 Contributions

This work illustrates the implementation of a domain-specific language utilizing three language workbenches and three metaprogramming languages. The

representative selection is based on diversity in the used paradigms. Also a general view, in the form of feature models, based on the features used by the implementations is given.

Therefore this thesis makes the following contributions:

- An implementation of the FSML using the language workbenches:
  - MPS
  - Rascal
  - Spoofax
- An implementation of the FSML using the metaprogramming languages:
  - Haskell
  - Racket
  - Scala
- Feature models based on the utilized features for implementing the FSML per representative.
- An accumulated model for features provided by language workbenches and metaprogramming languages.

Chapter 2

Related Work

Comparing implementations of domain-specific languages utilizing language workbenches has been done by the Language Workbench Challenge. Over three years, each year, the participants were challenged to realize a given domain-specific language with their language workbench [3]

Domain-specific languages were also implemented with the Rascal language workbench in [19] and the Spoofax language workbench in [4]. A typed version on top of Racket was developed using Racket itself in [21]. Other research in implementing languages, with Template Haskell, is described in [7]. Extensions for the Java language are implemented with MPS in [11].

Chapter 3

Implementation

This chapter is designated to the implementation of the FSML. Every implementation tries to be as close as possible to the one done by Lämmel in *Model of a DSL for finite state machines* [5]. The source code of the implementations can be found online [17].

3.1 MPS

3.1.1 Abstract Syntax

Structures represent abstract syntax tree nodes in MPS [11]. In Figure 3.1 the fsm structure is pictured. It extends the abstract context expression from the base language. This enables the fsm node to be used later as an Java expression. The other nodes are defined the same way.

```
concept Fsm extends    Expression
            implements <none>

  instance can be root: false
  alias: <no alias>
  short description: <no short description>

  properties:
  << ... >>

  children:
  states : State[0..n]

  references:
  << ... >>
```

Figure 3.1: Structure for the Fsm Node in MPS

3.1.2 Concrete Syntax

The concrete syntax for the FSML is defined by editors for the abstract syntax. This reflects the projectional nature of MPS [11]. In Figure 3.2 the cell layout contains the fsm keyword and a vertical collection of cells representing the states of the fsm.

```
<default> editor for concept Fsm
  node cell layout:
    fsm {
      [/
        <constant>
        (/ % states %                    /)
           /empty cell: <default>
      /]
    }

  inspected cell layout:
    <choose cell model>
```

Figure 3.2: Editor for the Fsm Structure in MPS

3.1.3 Constraints

Constraints with custom error or warning messages are built by defining non-typesystem rules [8]. The non-typesystem rule in Figure 3.3 applies to the fsm abstract syntax tree node. Each unreachable state node in the fsm a warning is generated.

```
checking rule check_Fsm {
  applicable for concept = Fsm as fsm
  overrides false

  do {
    fsm.getUnreachableStates().forEach({~it => warning "unreachable state " + it.name -> it; });
  }
}
```

Figure 3.3: Non-typesystem Rule for the Fsm Structure in MPS

3.1.4 Code Generation

The code generator uses reduction rules to transform the FSML structures to the MPS Base Language. Optionally reduction rules can be applied based on conditions. Ultimately the code generator uses the MPS Base Language to generate Java source code.

Figure 3.4 shows the reduction rule for the fsm node. The template on the right side represents the creation of a new `Fsm` object.

```
[concept     Fsm       ] --> <T  <TF [new Fsm(new linkedlist<State>{$COPY_SRCL$[null]}, new linkedlist<] TF>  T>
[inheritors false      ]              Transition>{$COPY_SRCL$[null]})
[condition  <always>]
```

Figure 3.4: Reduction Rule for the Fsm Structure in MPS

3.1.5 Tests

MPS provides a DSL for testing language implementations. Figure 3.5 illustrates the negative test case for the reachable state condition. This is done by indicating that a cell in the editor should contain an error or warning. In this case the `stateC` cell is expected to contain the `UnreachableState` warning.

```
Test case ReachabilityNotOk
nodes
  ( fsm {
      initial state stateA {
                      eventI / actionII -> stateB ;
                    }
      <no initial> state stateB {
                        << ... >>
                      }
      <check <no initial> state stateC {     has warning UnreachableState>
                            << ... >>
                          }
    } )
```

Figure 3.5: Tests in MPS

3.2 Rascal

3.2.1 Syntax Definition

In Rascal a language syntax is defined using context-free grammars. The modular definition allows extending and combining of languages [19].

Listing 3.1 shows the syntax definition of the FSML in Rascal. First the layout module from Rascals standard library is extended, which ensures whitespaces are skipped. Extending a module makes every definition local available for the extending module. The definition of `Fsm` indicates the start symbol, followed by defining the remaining nonterminals. Each syntax production rule can be labeled by an identifier as for every symbol in these production rules a postfix label can be defined. Further `Name` is defined as a lexical production rule.

The distinction between lexical and syntax rules is necessary due to the fact that Rascal interleaves the layout nonterminals with the syntax production rules before generating a parser. This step is skipped for lexical production rules so these production rules can contain character sequences which would otherwise be skipped by the parser.

```
1  extend lang::std::Layout;
2
3  start syntax Fsm = @Foldabel fsm: State* states;
4  syntax State = @Foldable state: "initial"? "state" Id id "{"
       ↪ Transition* transitions "}";
5  syntax Transition = transition: Input input ("/" Action
       ↪ action)? ("-\>" Id id)? ";";
6
7  syntax Id = Name;
8  syntax Input = Name;
9  syntax Action = Name;
10 lexical Name = ([a-zA-Z][a-zA-Z0-9]* !>> [a-zA-Z0-9]);
```

Listing 3.1: Syntax definition in Rascal

3.2.2 Constraint Checking

Constraints are checked in ordinary Rascal code. Each function returns a set of the `Message` algebraic data type from Rascals standard library. A `Message` is used to communicate information about source code. Respective a `Message` either represents an error, warning or info with the corresponding source location.

Single Initial State

The function in Listing 3.2 checks if the given finite-state machine has exactly one initial state. Otherwise an error message for every violation against the single initial state constraint will be returned. First the **`visit`** statement collects each initial and noninitial state in different sets. The following **`switch`** statement matches the set of initial states either with an empty set, a set with at least two elements or falls back to the default case.

```
1  private set[Message] checkSingleInitial(Fsm f) {
2    set[State] initialStates = {};
3    set[State] noninitialStates = {};
4    set[Message] messages = {};
5    visit(f) {
6      case state: (State)`initial state <Id _> { <Transition* _>
     ↪ }`: initialStates += state;
7      case state: (State)`state <Id _> { <Transition* _> }`:
     ↪ noninitialStates += state;
8    }
9    switch(initialStates) {
10     case {}: messages = {error("no initial state defined",
     ↪ n@\loc) | n <- noninitialStates};
11     case {X, Y, N*}: messages = {error("multiple initial
     ↪ states defined", i@\loc) | i <- initialStates};
12     default: el = {};
13   }
14   return messages;
15 }
```

Listing 3.2: Check for single initial state constraint in Rascal

Distinct State IDs

As shown in Listing 3.3 error messages for already defined state IDs are created by first collecting each defined state ID and then filter for duplicates.

```
1  private set[Message] checkDistinctIds(Fsm f) {
2    list[Id] ids = [ s.id | s <- f.states ];
3    return {error("state with ID <id> already defined", id@\loc)
       ↪ | id <- getDuplicates(ids)};
4  }
```

Listing 3.3: Check for distinct state IDs constraint in Rascal

Resolvable State ID

Unresolvable state IDs are found by building the relative complement of the set of defined state IDs in the set of referenced state IDs as show in Listing 3.4.

```
1  private set[Message] checkResolvable(Fsm f) {
2    set[Id] referencedIds = {};
3    set[Id] stateIds = {};
4    visit(f) {
5      case State s: stateIds += s.id;
6      case (Transition)`<Input _> / <Action _> -\> <Id id>;`:
       ↪ referencedIds += id;
7      case (Transition)`<Input _> -\> <Id id>;`: referencedIds
       ↪ += id;
8    }
9    return {error("unresolved state <id>", id@\loc) | id <-
       ↪ referencedIds - stateIds};
10 }
```

Listing 3.4: Check for resolvable state ID constraint in Rascal

Deterministic Input

The Listing 3.5 shows how state machines are checked for nondeterministic transitions. This is done by nesting two visit statements. First the outer one builds for every state a new list of inputs. Then the inner one fills its list of inputs and adds for every duplicated entry in this list an error message to the list of messages.

```
1  private set[Message] checkStateDeterministic(Fsm f) {
2    set[Message] messages = {};
3    visit(f) {
4      case State s: {
5        list[Input] inputs = [];
6        visit(s.transitions) {
7          case Input i: inputs += i;
8        };
9        messages += ({error("input <i> already defined in state
       ↪ <s.id>", i@\loc) | i <- getDuplicates(inputs)});
10     }
11   };
12   return messages;
13 }
```

Listing 3.5: Check for deterministic input constraint in Rascal

Reachable State

Rascals type **`rel`** is a shorthand for a set of tuples, where each tuple has the same static tuple type. To compute the composition of binary relations, Rascals operator **`o`** is used.

The function shown in Listing 3.6 returns a set of warning messages for each state not reachable from the initial state by any concatenation of valid

transitions. In its body first two sets of binary relations are build, one for the transitions of the initial state and the other one for the transitions of the remaining states. The tuples first element contains the transitions enclosing state and the second element is the transitions target state. Then Rascals **solve** statement resolves the fixed point computation. Where the fixed point is the

```
1  private set[Message] checkReachable(Fsm f) {
2    rel[State, State] initial = {};
3    rel[State, State] relation = {};
4    visit(f) {
5      case state: (State)`initial state <Id _> {<Transition*
     ↪ ts>}`:
6        initial += <state, state> + getStateRelation(f, state);
7      case state: (State)`state <Id _> {<Transition* ts>}`:
8        relation += getStateRelation(f, state);
9    }
10   solve(initial) {
11       initial += (initial o relation);
12   }
13   set[State] unreachableStates = {s | s <- f.states} - {s |
     ↪ <_, s> <- initial};
14   return {warning("unreachable state <s.id>", s.id@\loc) | s
     ↪ <- unreachableStates};
15 }
```

Listing 3.6: Check for reachable state constraint in Rascal

3.2.3 Code Generation

A code generator is defined using Rascals string templates. Strings can span over more than one line and a single quote indicates which part of the line is ignored. Expressions interpolated into strings are escaped between angle brackets. The result of interpolated expressions are on the same indentation level as the expression itself.

In Listing 3.7 a generator targeting Java source code is shown. For the source code generation of transitions parametric polymorphism is used for the different cases of transitions.

```
1  private str fsmToJava((Fsm)`<State* states>`, loc location) =
2    "package org.softlang.fluent;
3    '
4    'import static org.softlang.fluent.FsmlImpl.fsm;
5    '
6    'public class <getFileName(location)> {
7    '
8    ' public static final Fsml
     ↪ <toUpperCase(getFileName(location))> =
9    '    fsm()
10   '      <intercalate("\n", [stateToJava(s) | s <- states])>;
11   '}
12   ";
13
```

```
14 private str stateToJava(State state) =
15   ".state(\"<state.id>\")
16   ' <intercalate("\n", [transitionToJava(t) | t <-
     ↪ state.transitions])>";
17
18 private str transitionToJava((Transition)`<Input input>;`) =
19   ".transition(\"<input>\", null, null)";
20
21 private str transitionToJava((Transition)`<Input input> /
     ↪ <Action action>;`) =
22   ".transition(\"<input>\", \"<action>\", null)";
23
24 private str transitionToJava((Transition)`<Input input> -\>
     ↪ <Id id>;`) =
25   ".transition(\"<input>\", null, \"<id>\")";
26
27 private str transitionToJava((Transition)`<Input input> /
     ↪ <Action action> -\> <Id id>;`) =
28   ".transition(\"<input>\", \"<action>\", \"<id>\")";
```

Listing 3.7: Code generation in Rascal

3.2.4 Pretty Printing

The pretty printing functions defined in Listing 3.8 are very similar to the source code generation functions shown in subsection 3.2.3. They also utilize Rascals string interpolation capabilities.

```
1
2  private str ppFsm((Fsm)`<State* states>`) =
3    "<intercalate("\n", [ppState(s) | s <- states])>";
4
5  private str ppState((State)`initial state <Id id> {
     ↪ <Transition* transitions>}`) =
6    "initial state <id> {
7    ' <intercalate("\n", [ppTransition(t) | t <- transitions])>
8    '}
9    '";
10
11 private str ppState((State)`state <Id id> { <Transition*
     ↪ transitions> }`) =
12   "state <id> {
13   ' <intercalate("\n", [ppTransition(t) | t <- transitions])>
14   '}
15   '";
16
17 private str ppTransition((Transition)`<Input input>;`) =
18   "<input>;";
19
20 private str ppTransition((Transition)`<Input input> / <Action
     ↪ action>;`) =
21   "<input> / <action>;";
22
23 private str ppTransition((Transition)`<Input input> -\> <Id
     ↪ id>;`) =
24   "<input> -\> <id>;";
```

```
25
26 private str ppTransition((Transition)`<Input input> / <Action
     ↪ action> -\> <Id id>;`) =
27   "<input> / <action> -\> <id>;";
```

Listing 3.8: Pretty printing in Rascal

3.2.5 IDE Construction

Rascal integrates with Eclipse trough the IMP framework [19].

Listing 3.9 shows how the FSML is registered with the IMP runtime. First the language is registered with a name, file name extension and parser. Then the contributions for that language are registered. These contributions consist of a popup menu, annotator and outliner.

```
1  private set[Contribution] FSMLContrib = {
2    popup(
3      menu("FSML", [
4        action("Format", ppFsm),
5        action("Generate Java", generateJava),
6        action("Visualize", visualize)
7      ])
8    ),
9    annotator(annotateFsm),
10   outliner(outlineFsm)
11 };
12
13 public void registerFSML() {
14   registerLanguage(FSML_NAME, FSML_EXT, parseFsm);
15   registerContributions(FSML_NAME, FSMLContrib);
16 }
```

Listing 3.9: Plugin

Annotations

Listing 3.10 shows the underlying function of the registered annotater.

For each `Message` returned by the constraint checker an error marker is set in the IDE. The resolve function adds a hyperlink from the target state of an transition to the location where the state is defined.

```
1  public Fsm annotateFsm (Fsm f) {
2    set[Message] errors = checkConstraints(f);
3    f = resolve(f);
4    return f[@messages = errors];
5  }
6
7  private Fsm resolve(Fsm f) {
8    map[str, loc] stateIds = ();
9    map[str, loc] transitionIds = ();
10   visit(f) {
11     case State s: stateIds["<s.id>"] = s.id@\loc;
12   }
```

```
13    return visit(f){
14      case Transition t => visit(t) {
15        case Id id => id[@link=stateIds["<id>"]]
16          when "<id>" in stateIds
17      }
18    }
19  }
```

Listing 3.10: Annotations

Outline

The first function in Listing 3.11 returns a tree-like structure. Where the roots childs are the states of the finite state machine and the leaves of those childs are the corresponding transitions. A node is labeled by the state ID or the transition input. Also an reference to the source code location is set.

```
1   public node outlineFsm(Fsm f){
2     node outline = "states"([outlineState(s) | s <- f.states]);
3     return outline;
4   }
5
6   private node outlineState(State s) {
7     node outline = "<s.id>"([outlineTransition(t) | t <-
        ↪ s.transitions]);
8     outline@\loc = s@\loc;
9     return outline;
10  }
11
12  private node outlineTransition(Transition t) {
13    node outline = "<t.input>"();
14    outline@\loc = t@\loc;
15    return outline;
16  }
```

Listing 3.11: Outline

3.2.6 Tests

Tests are ordinary parameterless boolean functions prefixed by the modifier **test**. They can be called by every other function or by the unit test framework, by typing `:test` at the command line.

The test functions shown in Listing 3.12 succeed if the negative test cases produce the corresponding errors or warnings.

```
1   test bool initialNotOk2() {
2     Fsm fsm = parse(#Fsm,
        ↪ |project://fsml/src/main/resources/initialNotOk1.fsml|);
3     return {H*, error("no initial state defined", L), T*} :=
        ↪ checkConstraints(fsm);
4   }
5
6   test bool initialNotOk2() {
```

```
7    Fsm fsm = parse(#Fsm,
      ↪ |project://fsml/src/main/resources/initialNotOk2.fsml|);
8    return {H*, error("multiple initial states defined", L), T*}
      ↪ := checkConstraints(fsm);
9  }
10
11 test bool idsNotOk() {
12   Fsm fsm = parse(#Fsm,
      ↪ |project://fsml/src/main/resources/idsNotOk.fsml|);
13   return {H*, error("state with ID stateA already defined",
      ↪ L), T*} := checkConstraints(fsm);
14 }
15
16 test bool resolutionNotOk() {
17   Fsm fsm = parse(#Fsm,
      ↪ |project://fsml/src/main/resources/resolutionNotOk.fsml|);
18   return {H*, error("unresolved state stateC", L), T*} :=
      ↪ checkConstraints(fsm);
19 }
20
21 test bool determinismNotOk() {
22   Fsm fsm = parse(#Fsm,
      ↪ |project://fsml/src/main/resources/determinismNotOk.fsml|);
23   return {H*, error("input eventI already defined in state
      ↪ stateA", L), T*} := checkConstraints(fsm);
24 }
25
26 test bool reachabilityNotOk() {
27   Fsm fsm = parse(#Fsm,
      ↪ |project://fsml/src/main/resources/reachabilityNotOk.fsml|);
28   return {H*, warning("unreachable state stateC", L), T*} :=
      ↪ checkConstraints(fsm);
```

Listing 3.12: Tests in Rascal

3.3 Spoofax

3.3.1 Syntax Definition

Spoofax uses the Syntax Definition Formalism SDF3 for defining a context-free syntax. Therefore Listing 3.13 is a straightforward implementation of the context-free syntax of the FSML illustrated in *Model of a DSL for finite state machines* [5].

```
1  context-free start-symbols
2
3    Fsm
4
5  context-free syntax
6
7  Fsm.Fsm = State *
8
9  State.State = <<Initial ?> state <ID> {
10   <Transition *>
```

```
11 }>
12
13 Transition.Transition = <<Input> <Action ?> <Target ?>;>
14
15 Initial.Initial = <initial>
16
17 Input.Input = <<ID>>
18 Action.Action = </ <ID>>
19 Target.Target = <-\> <ID>>
```

Listing 3.13: Syntax definition in Spoofax

3.3.2 Name Binding

The NaBL Name Binding Language describes the scope and binding of syntax definitions.

In Listing 3.14 each Fsm node scopes its underlying state nodes. Theses state nodes define a unique ID in the Fsm scope. In the same way every transition node is scoped by a parent state node. At last the binding between the targets ID and the states ID is made.

```
1  namespaces
2    Fsm State Transition Input
3
4  binding rules
5
6    Fsm(_):
7      scopes State
8
9    State(_, id, _):
10     defines unique State id
11     scopes Transition
12
13   Transition(input, _, _):
14     defines unique Transition input
15
16   Target(id):
17     refers to State id
```

Listing 3.14: Name Binding in Spoofax

3.3.3 Constraint Checking

Constraints are checked by defining conditional rewrite rules in the Stratego language. Matching rules return a tuple of AST nodes and an associated message. Spoofax provides rules for constraint errors, warnings and notes.

Since the name binding ensures that the distinct ID, deterministic input and resolvable state constraints are satisfied, the AST is only checked for the single initial state and reachable state constraint.

Single Initial State

Listing 3.15 shows rules for constraint errors. The first rule is applied when no initial states are defined. Where the second rule is applied when there are more than one initial states.

```
1      Fsm(states*) ->
2        (states*, $[no initial state defined])
3        with
4          s* := <initial-states> states*;
5          n := <length> s*
6        where
7          <lt> (n, 1)
8
9    constraint-error:
10     Fsm(states*) ->
11       (initial-state*, $[multiple initial states defined])
12       with
13         initial-state* := <initial-states> states*;
14         n := <length> initial-state*
15       where
16         <gt> (n, 1)
```

Listing 3.15: Check for single initial state constraint in Spoofax

Reachable State

The constraint warning rule in Listing 3.16, which checks for unreachable states, is always applied. When there are no unreachable states the tuples first value contains an empty list.

```
1
2    constraint-note:
3      _ -> <fail>
4
5    constraint-warning:
6      Fsm(state*) ->
```

Listing 3.16: Check for reachable state constraint in Spoofax

3.3.4 Code Generation

Spoofax uses rewrite rules and string interpolation written in the Stratego transformation language to generate arbitrary source code. Listing 3.17 defines the entry point for Java source code generation.

```
1    to-java:
2        Fsm(state*) ->
3        $[
4          package de.sschauss.fsml;
5
6          public class Sample {
```

```
7
8           public static Fsm fsm;
9
10          static {
11            fsm = new Fsm();
12
13            [state'*]
14        }
15
16        }
17      ]
18    with
19          state'* := <to-java> state*
```

Listing 3.17: Code generation in Spoofax

3.3.5 IDE Construction

3.3.6 Tests

Test are written in the Spoofax Testing Language SPT.

In Listing 3.18 negative test cases for the FSML are shown. Each test statement expects an exact number of warnings or errors.

```
1  test Determinisim not ok [[
2    initial state stateA {
3        eventI / actionI -> stateB;
4        eventI / actionII -> stateC;
5    }
6    state stateB { }
7    state stateC { }
8  ]] 2 error
9
10 test Reachability not ok [[
11   initial state stateA {
12       eventI/actionI -> stateB;
13   }
14   state stateB { }
15   state stateC { }
16 ]] 1 warning
```

Listing 3.18: Tests in Spoofax

3.4 Haskell

3.4.1 Abstract Syntax Tree

Haskells algebraic data types are predestinated to represent the nodes of the abstract syntax tree. Listing 3.19 shows a straight forward implementation using these data types.

```
1  module Language.Fsml.AST where
```

```
2
3  data Fsm = Fsm  { states :: [State] }
4
5  data State = State { initial     :: Bool
6                     , id          :: String
7                     , transitions :: [Transition] }
8                     deriving (Eq)
9
10 data Transition = Transition { input  :: String
11                              , action :: Maybe String
12                              , target ::  Maybe String }
13                              deriving (Eq)
```

Listing 3.19: Abstract syntax tree in Haskell

3.4.2 Concrete Syntax

The concrete syntax realized in Listing 3.20 also uses algebraic data types. This implementation differs from the abstract syntax tree representation by using the states type for the target state in the transitions algebraic data type instead of a string.

```
1  module Language.Fsml.CS where
2
3  data Fsm = Fsm { states :: [State] } deriving (Eq, Show)
4
5  data State = State { initial     :: Bool
6                     , id          :: String
7                     , transitions :: [Transition] }
8                     deriving (Eq, Show)
9
10 data Transition = Transition { input  :: String
11                              , action :: Maybe String
12                              , target :: Maybe State }
13                              deriving (Eq, Show)
```

Listing 3.20: Concrete syntax tree in Haskell

3.4.3 Parser

Listing 3.21 shows the parser for the FSML. The parser is implemented using the parser combinator library Parsec [10]. Ultimately the `fsm` function transforms FSML source code to an abstract syntax tree representation.

```
1  fsmlDef :: Token.LanguageDef ()
2  fsmlDef = emptyDef
3      { Token.commentStart     = "/*"
4      , Token.commentEnd       = "*/"
5      , Token.commentLine      = "//"
6      , Token.identStart       = lower
7      , Token.identLetter      = letter
8      , Token.nestedComments   = True
9      , Token.reservedNames    = ["initial", "state"]
```

```
10      , Token.reservedOpNames  = ["=", "/", "->"]
11      }
12
13  lexer :: Token.TokenParser ()
14  lexer = Token.makeTokenParser fsmlDef
15
16  braces :: Parser p -> Parser p
17  braces = Token.braces lexer
18
19  semi :: Parser String
20  semi = Token.semi lexer
21
22  reserved :: String -> Parser ()
23  reserved = Token.reserved lexer
24
25  reservedOp :: String -> Parser ()
26  reservedOp = Token.reservedOp lexer
27
28  symbol :: String -> Parser String
29  symbol = Token.symbol lexer
30
31  topLevel :: Parser p -> Parser p
32  topLevel p = spaces *> p <* spaces
33
34  identifier :: Parser String
35  identifier = Token.identifier lexer
36
37  initial :: Parser Bool
38  initial =
39          (symbol "initial" >> return True)
40      <|> (symbol ""        >> return False)
41
42
43  transition :: Parser AST.Transition
44  transition = AST.Transition <$> identifier <*> optionMaybe
        ↪ (reservedOp "/" *> identifier) <*> optionMaybe
        ↪ (reservedOp "->" *> identifier) <* semi
45
46  state :: Parser AST.State
47  state = AST.State  <$> initial <*> (reserved "state" *>
        ↪ identifier) <*> braces (many transition)
48
49  fsm :: Parser AST.Fsm
50  fsm = AST.Fsm <$> many state
```

Listing 3.21: Parser in Haskell

3.4.4 Quasiquotation

The Template Haskell library provides quasiquotation capabilities to Haskell [20].

In Listing 3.22 the quoter for the FSML is defined. Pattern, type and declaration quotation is omitted, as the quoted FSML source code is only expected in an expression context.

The function `quoteFsmlExp` invokes the parser and constraint checker. When these steps succeed, the abstract syntax tree is passed to the `toFsmLetExp`.

This is where the abstract syntax tree is transformed to Template Haskells quotation monad. Quotation monads encapsulate the concept of generating fresh names, failure, input and output [18].

```
1  fsml :: QuasiQuoter
2  fsml = QuasiQuoter
3      { quoteExp   = quoteFsmlExp
4      , quotePat   = undefined
5      , quoteType  = undefined
6      , quoteDec   = undefined
7      }
8
9  quoteFsmlExp :: String -> Q Exp
10 quoteFsmlExp str = do
11     filename <- loc_filename `fmap` location
12     case parse (topLevel fsm) filename str of
13         Left err  -> error (show err)
14         Right ast -> either (error . show) toFsmLetExp (check
   ↪ ast)
15
16 toFsmLetExp :: AST.Fsm -> Q Exp
17 toFsmLetExp fsmNode = [|$(letE (map toStateDec (AST.states
   ↪ fsmNode)) (toFsmExp fsmNode))|]
18
19 toFsmExp :: AST.Fsm -> Q Exp
20 toFsmExp (AST.Fsm states) = [|CS.Fsm $(listE (map (varE .
   ↪ mkName . AST.id) states))|]
21
22 toStateDec :: AST.State -> Q Dec
23 toStateDec state = valD (varP (mkName (AST.id state)))
   ↪ (normalB (toStateExp state)) []
24
25 toStateExp :: AST.State -> Q Exp
26 toStateExp (AST.State initial id transitions) = [|CS.State
   ↪ initial id $(listE (map toTransitionExp transitions))|]
27
28 toTransitionExp :: AST.Transition -> Q Exp
29 toTransitionExp (AST.Transition input action target) =
   ↪ [|CS.Transition input action $(maybe (conE 'Nothing)
   ↪ (appE (conE 'Just) . varE . mkName) target)|]
```

Listing 3.22: Quoter in Haskell

3.4.5 Constraint Checking

The checks for distinct IDs and resolvable state IDs are not implemented as the Haskell compiler already takes care of unique and resolvable names. A finite state machine is considered valid if every of the following function returns an empty list.

Single Initial State

Listing 3.23 uses `initialStates` to select the initial states. If there are no initial states or more than one, an appropriate error message is returned.

```haskell
checkSingleInitial :: [AST.State] -> [String]
checkSingleInitial states =
    case length (initialStates states) of
        0   -> ["No initial state defined"]
        1   -> []
        _   -> ["Multiple initial states defined"]
```

Listing 3.23: Check for single initial state constraint in Haskell

Deterministic Input

Checking for the deterministic input constraint is done in Listing 3.24. The first function concats the list of the duplicated inputs found in each states by invoking the second function with every state.

```haskell
checkDeterminism :: [AST.State] -> [String]
checkDeterminism = concatMap (checkDeterminism' .
    ↪ AST.transitions)
checkDeterminism' :: [AST.Transition] -> [String]
checkDeterminism' = dup . map ((++) "Duplicated input " .
    ↪ AST.input)
```

Listing 3.24: Check for deterministic input constraint in Haskell

Reachable State

Listing 3.25 uses the initial states as an entry point to find all reachable states. Then `checkReachability'` determines which states to visit by selecting every target state referenced in the transitions of the currently states to visit. This functions invokes itself as long as new target states are found.

```haskell
checkReachability :: [AST.State] -> [String]
checkReachability states =
    case checkReachability' states (initialStates states)
    ↪ (initialStates states) of
        [] -> []
        s  -> map ("Unreachable state " ++) s

checkReachability' ::  [AST.State] -> [AST.State] ->
    ↪ [AST.State] -> [String]
checkReachability' s visited [] = map AST.id s \\ map AST.id
    ↪ visited
checkReachability' s visited toVisit = checkReachability' s
    ↪ visited' toVisit'
    where
        visited' = visited ++ toVisit
        toVisit' = (map (\i -> targetState i s) (mapMaybe
    ↪ AST.target (concatMap AST.transitions toVisit))) \\
    ↪ visited'
```

Listing 3.25: Check for reachable state constraint in Haskell

3.4.6 Tests

As the quasiquoter runs when the program is compiled, the doctest library is used to test for compile-time errors [2]. This is possible due to the fact that each test case defined in a Haddock comment is evaluated by a read-eval-print loop like ghci.

An example negative is test case is defined in Listing 3.26

```
-- | >>> :set -XQuasiQuotes
{- | >>> :{
    [fsml|
        initial state stateA {
            eventI / actionI -> stateB;
            eventI / actionII -> stateC;
        }
        state stateB {
        }
        state stateC {
        }
    |]
:}
...
...Duplicated input eventI...
...
-}
```

Listing 3.26: Tests in Haskell

3.5 Racket

3.5.1 Concrete Syntax

Struct types are used in Listing 3.27 to define the concrete syntax of the FSML in Racket.

```
(struct fsm-struct (states))
(struct state-struct (initial id transitions))
(struct transition-struct (input action target))
```

Listing 3.27: Macros in Racket

3.5.2 Macros

The first macro defined in Listing 3.28 extracts the name and states from the syntax object. Then the `with-syntax` form bind the identifier to a syntax object representing the state IDs. After that the input syntax object is checked for violation constraints (subsection 3.5.4). Finally a begin form syntax object is returned, where the states and the `fsm-struct` definition are spliced in.

States are transformed to define forms syntax object by the second and third macro definition.

```
1  (define-syntax (fsml stx)
2    (syntax-case stx ()
3      [(_ name state ...)
4       (with-syntax ([(state-ids ...) (stx-map state->id
   ↪ #'(state ...))])
5       (check-fsml stx)
6       #'(begin state ... (define name (fsm-struct (list
   ↪ state-ids ...)))))]))
7
8  (define-syntax (initial stx)
9    (syntax-case stx ()
10     [(_ . rest)
11      (with-syntax ([state (stx->define-state-struct #'rest
   ↪ #t)])
12        #'state)]))
13
14 (define-syntax (state stx)
15   (stx->define-state-struct stx))
```

Listing 3.28: Macros in Racket

3.5.3 Syntax Object Transformation

Each forms phase level in Listing 3.29 is shifted by the `begin-for-syntax` form by one. This is required, as these forms are used by the macros at compile time. The `state->id` procedure transforms a state syntax object to an ID syntax object. A transition syntax object is transformed by the `stx->transition-struct` procedure to a `transition-struct` syntax object. Finally the `stx->define-state-struct` function converts a state syntax object to a syntax object representing a define form of a `state-struct`.

```
1  (begin-for-syntax
2
3    (define (state->id stx)
4      (syntax-case stx (initial state)
5        [(initial state id { transition ... })
6         #'id]
7        [(state id { transition ... })
8         #'id]))
9
10   (define (stx->transition-struct stx)
11     (syntax-case stx (/ ->)
12       [(input)
13        (with-syntax ([input-string (id->string input)])
14          #'(transition-struct input-string #f #f))]
15       [(input / action)
16        (with-syntax ([input-string (id->string input)]
17                      [action-string (id->string action)])
18          #'(transition-struct input-string action-string #f))]
19       [(input -> target)
20        (with-syntax ([input-string (id->string input)])
21          #'(transition-struct input-string #f (λ () target)))]
22       [(input / action -> target)
23        (with-syntax ([input-string (id->string input)]
```

```
24                       [action-string (id->string action)])
25          #'(transition-struct input-string action-string (λ ()
   ↪ target)))]))
26
27  (define stx->define-state-struct
28    (λ (stx [initial? #f])
29      (syntax-case stx ()
30        [(_ id { transition-stx ... })
31         (with-syntax ([id-string (id->string id)]
32                       [initial? initial?]
33                       [(transition ...) (stx-map
   ↪ stx->transition-struct #'(transition-stx ...))])
34           #'(define id (state-struct initial? id-string (list
   ↪ transition ...))))])))
```

Listing 3.29: Syntax Object Transformation in Racket

3.5.4 Constraint Checking

As for every state a define form is generated and the target in a transition has a binding for a state identifier, the constraint checks for the distinct state ID and resolvable state ID constraints don't have to be checked for.

The validation procedures for deterministic input and reachable state are omitted, because all checks are implemented similar.

Single Initial State

The function within Listing 3.30 uses the syntactic form `raise-syntax-error` to indicate that a syntax error has occurred.

```
1  (define (check-fsml-single-initial stx)
2    (let ([initial-states (stx->initial-states stx)])
3        (case (length initial-states)
4          [(0) (raise-syntax-error 'constraint-error
5                                   "no initial state defined"
6                                   stx)]
7          [(1) (void)]
8          [else (raise-syntax-error 'constraint-error
9                                   "initial state already
   ↪ defined"
10                                  (second initial-states))])))
```

Listing 3.30: Check for single initial state constraint in Racket

3.5.5 Tests

Racket ships with the unit-testing framework RackUnit, which is used for the implementation of the test cases [12]. Since these test cases bear a strong resemblance to each other, the tests for the single initial state constraint and reachable state constraint are omitted.

Test Helper

The procedure `syntax-error?` in Listing 3.31 is used to test if a given quoted syntax object, when evaluated, raises a syntax error. This is done by evaluating the result of a quasiquoted begin macro where the quoted syntax object is spliced in. If the evaluation raises a syntax error, the errors message is returned.

```
1 (define-namespace-anchor fsml-tests)
2 (define fsml-tests-ns (namespace-anchor->namespace fsml-tests))
3 (define (syntax-error? quoted-syntax)
4   (with-handlers ([exn:fail:syntax?
5                    (lambda (e) (exn-message e))]
6                   [exn:fail?
7                    (lambda (_) #f)])
8     (eval `(begin ,quoted-syntax #f) fsml-tests-ns)))
```

Listing 3.31: Test helper in Racket

Test Case

Test cases are build using the `test-case` form. The test case in Listing 3.32 checks that the regular expression matches the message returned by `syntax-error?`.

```
1  (test-case
2   "determinism constraint"
3   (check-regexp-match
4    "input already defined in state"
5    (syntax-error? '(fsml determinism-not-ok
6                          (initial state stateA {
7                                                 (eventI /
     ↪ actionI -> stateB)
8                                                 (eventI /
     ↪ actionII -> stateC)
9                                                 })
10                         (state B { })
11                         (state C { })))))
```

Listing 3.32: Test case in Racket

3.6 Scala

3.6.1 Abstract Syntax

The abstract syntax is represented by Scala case classes shown in Listing 3.33, Listing 3.34 and Listing 3.35.

```
1 case class FsmNode(states: List[StateNode])
```

Listing 3.33: Fsm Abstract Syntax node in Scala

```
1  case class StateNode(initial: Boolean, id: String,
   ↪ transitions: List[TransitionNode])
```

Listing 3.34: State Abstract Syntax node in Scala

```
1  case class TransitionNode(input: String, action:
   ↪ Option[String], target: Option[String])
```

Listing 3.35: Transition Abstact Syntax Node in Scala

3.6.2 Macro

Macro Annotation

Listing 3.36 shows the definition for the macro annotation. This is a feature of the Scala macro paradise plugin [6]. A macro annotation is an ordinary Scala class extending the `StaticAnnotation` trait. This trait expects a macro assigned to `macroTransform`. The `fsmMacro` variables refers to the actual implementation. To ensure that the annotation is not referred to after type checking, the class is annotated with the `compileTimeOnly` annotation.

```
1  @compileTimeOnly("macro annotation for the FSML")
2  class Fsm extends StaticAnnotation {
3    def macroTransform(annottees: Any*): Any = macro fsmMacro
4  }
```

Listing 3.36: Macro definition in Scala

Implicit Conversion

The implicit conversion in Listing 3.37 is used by the Scala compiler. When an expression of the type `Name` is used in place of the type `String`, the original expression is replaced by the invocation of the implicit conversion with the original expression as its parameter.

```
1      implicit val nameToString: Name => String = (ident: Name)
   ↪ => ident.toString
```

Listing 3.37: Implicit Conversion in Scala

Unlifting

The unlifting instances in Listing 3.38 transform Scala abstract syntax tree to the abstract syntax tree representation defined for the FSML in subsection 3.6.1. Pattern matching against a quasiquotation is used for the alternatives of a concrete syntax rule of the FSML.

```
1      implicit val unliftStateDefinition: Unliftable[StateNode]
    ↪ = Unliftable {
2        case q"initial state ${id: Name} { ..${transitions:
    ↪ List[TransitionNode]} }" =>
3          StateNode(true, id, transitions)
4        case q"state ${id: Name} { ..${transitions:
    ↪ List[TransitionNode]} }" =>
5          StateNode(false, id, transitions)
6      }
7
8      implicit val unliftTransitionExpression:
    ↪ Unliftable[TransitionNode] = Unliftable {
9        case q"${input: Name} / ${action: Name} -> ${target:
    ↪ Name}" =>
10         TransitionNode(input, Some(action), Some(target))
11       case q"${input: Name} / ${action: Name}" =>
12         TransitionNode(input, Some(action), None)
13       case q"${input: Name} -> ${target: Name}" =>
14         TransitionNode(input, None, Some(target))
15       case q"${input: Name}" =>
16         TransitionNode(input, None, None)
17     }
```

Listing 3.38: Unlifting in Scala

Lifting

Lifting instances shown in Listing 3.39 define the Scala abstract syntax tree representation for the state and transition data type. The first lifting instance transforms a `StateNode` to an object with the states ID as its name and the transitions as its body. A `TransitionNode` is lifted to a function declaration. The transition input is used for the functions name and the body contains a `println` invocation, if the transition contains an action. For transitions with a target state this state is returned by the function, otherwise the enclosing context is returned.

```
1      implicit val liftStateNode: Liftable[StateNode] = Liftable
    ↪ { s =>
2        q"""object ${TermName(s.id)} { ..${s.transitions} }"""
3      }
4
5      implicit val liftTransitionNode: Liftable[TransitionNode]
    ↪ = Liftable {
6        case TransitionNode(input, Some(action), Some(target)) =>
7          q"def ${TermName(s"$input")} = { println($action);
    ↪ ${TermName(target)} }"
8        case TransitionNode(input, Some(action), None) =>
9          q"def ${TermName(s"$input")} = { println($action);
    ↪ this }"
10       case TransitionNode(input, None, Some(target)) =>
11         q"def ${TermName(s"$input")} = ${TermName(target)}"
12       case TransitionNode(input, None, None) =>
13         q"def ${TermName(input)} = this"
14     }
```

Listing 3.39: Lifting in Scala

Tree Deconstruction

The Scala abstract syntax tree is deconstructed in Listing 3.40 by unquoting the annotees object name, parents and unquote splicing the objects body respective the states in a quasiquotation. For the unquote splicing of the states the unlifting instances defined in section 3.6.2 are used.

```
1       val q"object $objectName extends ..$parents {
    ↪ ..${states: List[StateNode]} }" = annottees.head
```

Listing 3.40: Tree Deconstruction in Scala

Tree Construction

In Listing 3.41 the resulting Scala abstract syntax tree is constructed defining a quasiquotation. The object name and parents are unquoted back in with the values the tree deconstruction extracted. Further in the objects body the initial state is unquoted in place of the right-hand site of the newly defined apply function. Finally the states are also spliced into the objects body. The lifting instances from section 3.6.2 are used to quote or splice the states into the tree.

```
1       q"""
2         object $objectName extends ..$parents {
3           def apply() =
    ↪ ${TermName(getInitialState(fsmNode).id)}
4           ..$states
5         }
6       """
```

Listing 3.41: Tree Construction in Scala

3.6.3 Constraint Checking

As the Scala compiler takes care of name resolution and name collision, there is no need to check for the distinct state ID constraint, resolvable state ID constraint and deterministic input constraint. In the following the single initial state constraint check is shown.

Single Initial State

Listing 3.42 function first counts the number of initial states. If the this result matches zero or more than one, a compile time error is raised.

```scala
private def checkSingleInitial(c: whitebox.Context)(fsm:
  ↪ FsmNode): Unit =
  fsm.states count {
    _.initial
  } match {
    case 0 => c.error(c.enclosingPosition, "No initial state
  ↪ defined")
    case 1 =>
    case _ => c.error(c.enclosingPosition, "Multiple initial
  ↪ states defined")
  }
```

Listing 3.42: Check for single initial state constraint in Scala

3.6.4 Tests

ScalaTest provides a convenient way to test if source code does not compile [16]. Listing 3.43 shows the negative test case for the single initial state constraint.

```scala
"A Fsm" should "not compile when determinism not Ok" in {
  """
  import de.sschauss.fsml.macros.Fsm
  @Fsm
  object DeterminismNotOk {
    initial state stateA {
      eventI / actionI -> stateB;
      eventI / actionII -> stateC;
    }
    state stateB { }
    state stateC { }
  }
  """ shouldNot compile
}
```

Listing 3.43: Tests in Scala

Chapter 4

Feature Models

This chapter gives an overview on features used by languages workbenches and general-purpose programming languages with meta-programming facilities to realize domain-specific languages. First these features are identified by analyzing the implementations of the FSML illustrated in chapter 3. Then feature models build on the identified features are constructed. Finally, the devised features models of the language workbenches respectively metaprogramming languages are combined.

4.1 Language Workbenches

4.1.1 MPS

MPS provides several domain-specific languages for implementing languages as seen in section 3.1. These domain-specific languages are in particular the structure language for abstract syntax tree definitions, editor language for concrete syntax definitions, type system language for constraint checking, generator language for code generation and test language [9]. It should be noted that the generator language uses reduction rules to generate code. The devised feature model is shown in Figure 4.1.

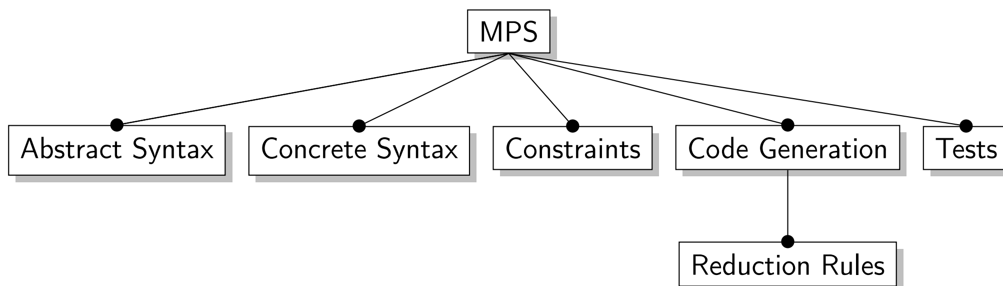

Figure 4.1: Feature Model for Haskell

4.1.2 Rascal

The implementation in section 3.2 of the FSML utilizing the Rascal language workbench used the following features. First, context-free grammars were

defined for the syntax definition. Then Rascals algebraic data type `Message` was provided for constraint checking purposes. Further string interpolation was used for source code generation. Finally the **`test`** modifier for testing. Every features was provided by the Rascal language itself [13]. Figure 4.2 shows the relevant feature model.

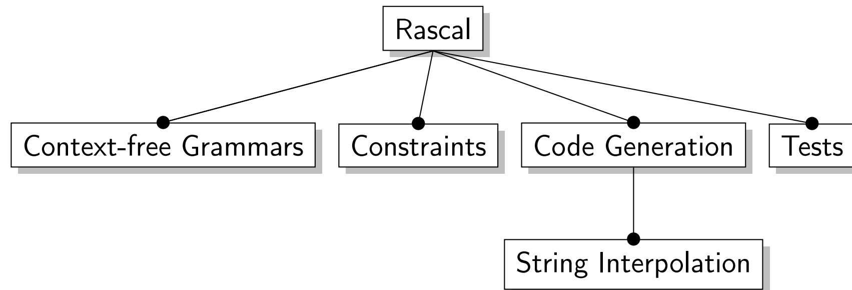

Figure 4.2: Feature Model for Rascal

4.1.3 Spoofax

The implementation in section 3.3 utilized four different domain-specific languages. First SDF3, the Syntax Definition Formalism, for the syntax definition of the FSML. Then NaBL, the Name Binding Language, for lexical scoping. Stratego program transformations for constraint checking and code generation. The code generation is implemented in the form of string interpolation. Finally, the Spoofax Testing Language is used to test the FSML implementation.

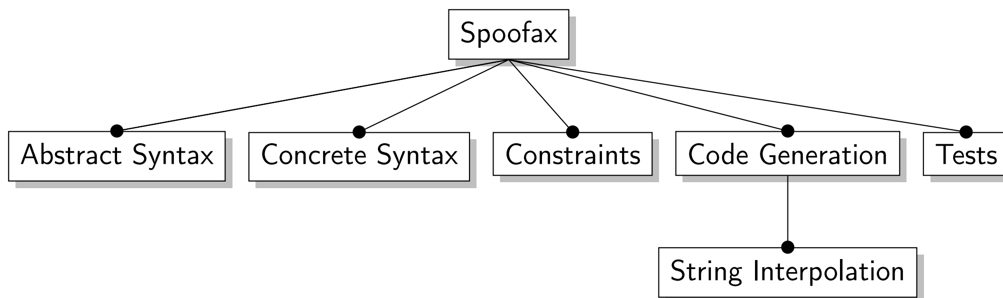

Figure 4.3: Feature Model for Spoofax

4.1.4 Combined Feature Model

The features identified for the language workbenches are combined in Figure 4.4. Every implementation used constraint checking, code generation and testing features of the language workbenches. Code generation was either realized using string interpolation or reduction rules.

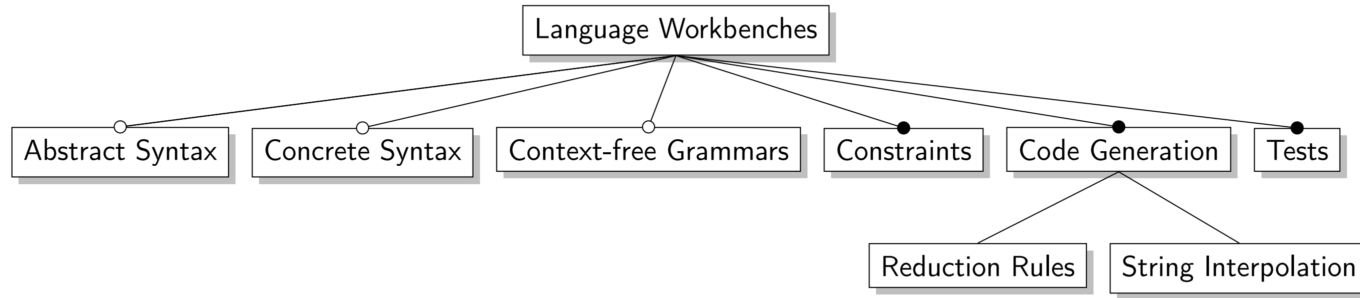

Figure 4.4: Feature Model for Language Workbenches

4.2 Metaprogramming Languages

4.2.1 Haskell

Haskells support for metaprogramming is achieved by using the Template Haskell library, as illustrated in section 3.4. Template Haskells quotation monad encapsulates the process of generating fresh names and lexical scoping [18]. Therefore some kind of hygiene is provided. A quasiquotation can occur instead of an expression, pattern, type or declaration context.

Template Haskell execution phase is limited to the compile time phase, as it is a compile time only meta-system [18]. Therefore the features are mapped to the feature model in Figure 4.5.

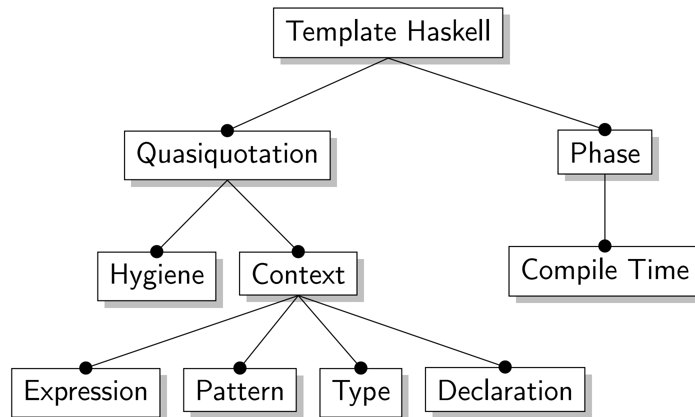

Figure 4.5: Feature Model for Haskell

4.2.2 Racket

Racket uses hygienic macros to provide support for metaprogramming [21]. The implementation of the FSML in section 3.5 showed macros at compile time and runtime. Quasiquotation in the expression context is used to manipulate syntax objects. Also the syntax transformations are bound at a different phase level than the macros. This implies that Racket allows the programmer to chose the stage procedures are evaluated. These features are pictured in Figure 4.6.

4.2.3 Scala

Scalas metaprogramming capabilities are supported by the Scala macro system [14] [1]. These capabilities are extended by the Scala macro paradise plugin. It makes features available for Scala that are not yet implemented in the current versions of Scala [6].

In section 3.6 quasiquotation was used to realize the implementation of the FSML. The quasiquotation syntax allows quoting in the context of expressions, types, patterns and auxiliary clauses [15]. Also lifting instances were provided to create abstract syntax tree representations for customs types, and unlifting

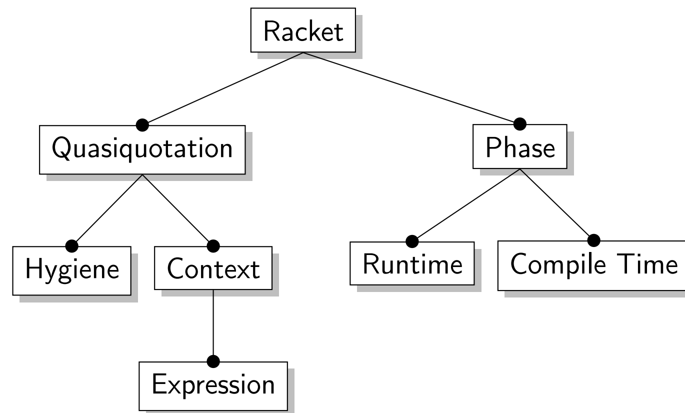

Figure 4.6: Feature Model for Racket

instances for the other way round. The implementations macro is expanded at compile time.

These features are pictured in Figure 4.7.

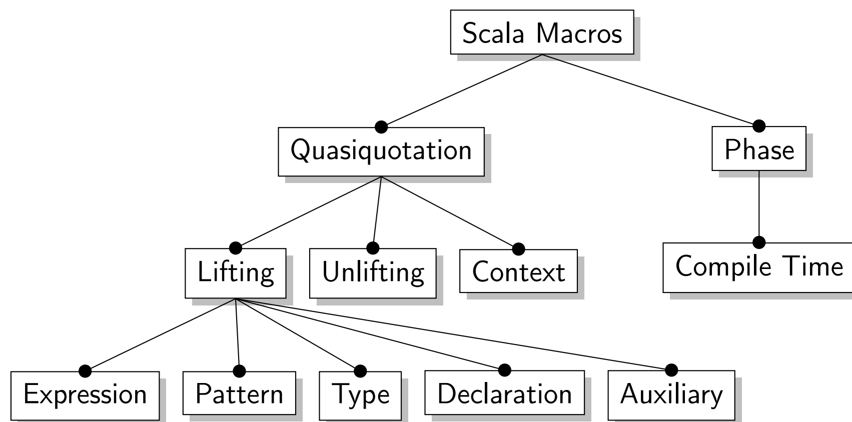

Figure 4.7: Feature Model for Scala

4.2.4 Combined Feature Model

Figure 4.8 shows the merged feature model from the models for Haskell, Racket and Scala. The only features used by all metaprogramming languages are quasiquotation in an expression context and metaprogramming at compile time.

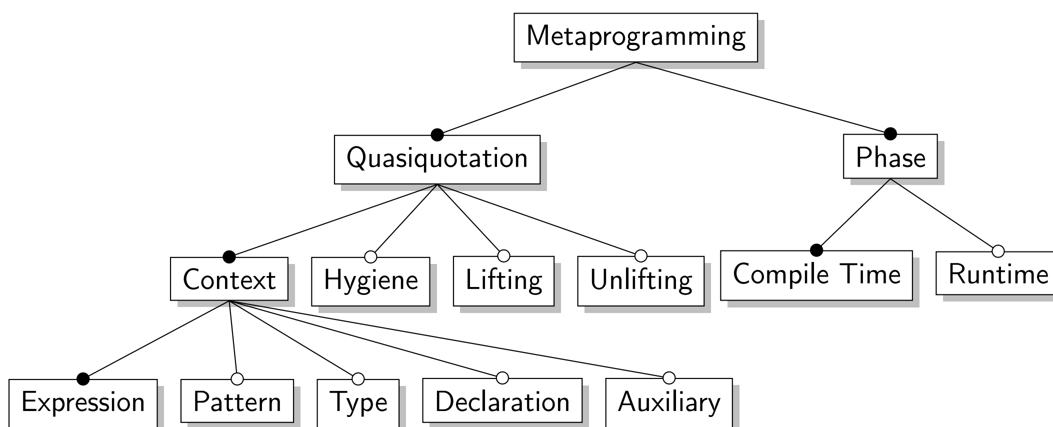

Figure 4.8: Feature Model for Metaprogramming

Chapter 5

Conclusion

This thesis illustrated how a domain-specific language like the FSML could be implemented using language workbenches and general-purpose programming languages with metaprogramming facilities. These implementations were used to explore the features provided by the selected representatives. Finally, these features were visualized in the form of feature models.

As the FSML has no type system, not every feature provided by the language workbenches and metaprogramming languages is represented in the devised feature models. On top of that the amount of representatives was quite small, which might also lead to inconclusive models.

Due to that, more features could be identified in the future by analyzing implementations of more sophisticated languages or implement languages utilizing other language workbenches and metaprogramming languages, or both.

Bibliography

[1] E. Burmako, “Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming”, in *Proceedings of the 4th Workshop on Scala*, ACM, 2013, p. 3.

[2] (). Doctest, [Online]. Available: `https://hackage.haskell.org/package/doctest` (visited on 03/15/2016).

[3] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, *et al.*, “The state of the art in language workbenches”, in *Software language engineering*, Springer, 2013, pp. 197–217.

[4] L. C. Kats and E. Visser, “The spoofax language workbench: Rules for declarative specification of languages and ides”, in *ACM sigplan notices*, ACM, vol. 45, 2010, pp. 444–463.

[5] R. Lämmel, *Model of a dsl for finite state machines*, 2014.

[6] (). Macro paradise, [Online]. Available: `http://docs.scala-lang.org/overviews/macros/paradise.html` (visited on 15/01/2016).

[7] G. Mainland, “Why it’s nice to be quoted: Quasiquoting for haskell”, in *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, ACM, 2007, pp. 73–82.

[8] (). Mps - common language patterns, [Online]. Available: `https://confluence.jetbrains.com/display/MPSD33/Common+language+patterns`.

[9] (). Mps documentation, [Online]. Available: `https://confluence.jetbrains.com/display/MPSD33/MPS+Useri's+Guide`.

[10] (). Parsec, [Online]. Available: `http://hackage.haskell.org/package/parsec` (visited on 03/14/2016).

[11] V. Pech, A. Shatalin, and M. Voelter, “Jetbrains mps as a tool for extending java”, in *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, ACM, 2013, pp. 165–168.

[12] (). Rackunit, [Online]. Available: `http://docs.racket-lang.org/rackunit/` (visited on 01/14/2016).

[13] (). Rascaltutor, [Online]. Available: `http://tutor.rascal-mpl.org/Rascal/Rascal.html` (visited on 11/01/2016).

[14] (). Scala macros, [Online]. Available: http://docs.scala-lang.org/overviews/macros/overview.html (visited on 15/01/2016).

[15] (). Scala quasiquotation, [Online]. Available: http://docs.scala-lang.org/overviews/quasiquotes/intro.html (visited on 15/01/2016).

[16] (). Scalatest, [Online]. Available: http://www.scalatest.org/ (visited on 01/03/2016).

[17] S. Schauß. (). Langtechs, [Online]. Available: https://github.com/softlangbook/langtechs (visited on 03/17/2016).

[18] T. Sheard and S. P. Jones, "Template meta-programming for haskell", in *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, ACM, 2002, pp. 1–16.

[19] T. van der Storm, *The Rascal language workbench*. CWI. Software Engineering [SEN], 2011.

[20] (). Template haskell, [Online]. Available: https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/template-haskell.html (visited on 27/02/2016).

[21] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen, "Languages as libraries", in *ACM SIGPLAN Notices*, ACM, vol. 46, 2011, pp. 132–141.