The Spoofax Language Workbench

Rules for Declarative Specification of Languages and IDEs

Lennart C. L. Kats
Delft University of Technology
l.c.l.kats@tudelft.nl

Eelco Visser
Delft University of Technology
visser@acm.org

Abstract

Spoofax is a language workbench for efficient, agile development of textual domain-specific languages with state-of-the-art IDE support. Spoofax integrates language processing techniques for parser generation, meta-programming, and IDE development into a single environment. It uses concise, declarative specifications for languages and IDE services. In this paper we describe the architecture of Spoofax and introduce idioms for high-level specifications of language semantics using rewrite rules, showing how analyses can be reused for transformations, code generation, and editor services such as error marking, reference resolving, and content completion. The implementation of these services is supported by language-parametric editor service classes that can be dynamically loaded by the Eclipse IDE, allowing new languages to be developed and used side-by-side in the same Eclipse environment.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.2.6 [*Software Engineering*]: Programming Environments

General Terms Languages

1. Introduction

Domain-specific languages (DSLs) provide high expressive power focused on a particular problem domain [38, 47]. They provide linguistic abstractions over common tasks within a domain, so that developers can concentrate on application logic rather than the accidental complexity of low-level implementation details. DSLs have a concise, domain-specific notation for common tasks in a domain, and allow reasoning at the level of these constructs. This allows them to be used for automated, domain-specific analysis, verification, optimization, parallelization, and transformation (AVOPT) [38].

For developers to be productive with DSLs, good integrated development environments (IDEs) for these languages are essential. Over the past four decades, IDEs have slowly risen from novelty tool status to becoming a fundamental part of software engineering. In early 2001, IntelliJ IDEA [42] revolutionized the IDE landscape [17] with an IDE for the Java language that parsed files as they were typed (with error recovery in case of syntax errors), performed semantic analysis in the background, and provided code navigation with a live view of the program outline, references to declarations of identifiers, content completion proposals as programmers were typing, and the ability to transform the program based on the abstract representation (refactorings). The now prominent Eclipse platform, and soon after, Visual Studio, quickly adopted these same features. No longer would programmers be satisfied with code editors that provided basic syntax highlighting and a "build" button. For new languages to become a success, state-of-the-art IDE support is now mandatory. For the production of DSLs this requirement is a particular problem, since these languages are often developed with much fewer resources than general purpose languages.

There are five key ingredients for the construction of a new domain-specific language. (1) A parser for the *syntax* of the language. (2) Semantic *analysis* to validate DSL programs according to some set of constraints. (3) *Transformations* manipulate DSL programs and can convert a high-level, technology-independent DSL specification to a lower-level program. (4) A *code generator* that emits executable code. (5) Integration of the language into an *IDE*.

Traditionally, a lot of effort was required for each of these ingredients. However, there are now many tools that support the various aspects of DSL development. Parser generators can automatically create a parsers from a grammar. Modern parser generators can construct efficient parsers that can be used in an interactive environment, supporting error recovery in case of syntax-incorrect or incomplete programs. Meta-programming languages [3, 10, 12, 20, 35] and frameworks [39, 57] make it much easier to specify the semantics of a language. Tools and frameworks for IDE development such as IMP [7, 8] and TMF [56], simplify the implementation of IDE services. Other tools, such as the Synthesizer

Generator [41], Centaur [2], and Lrc [37] can even generate a complete IDE from a syntactic (and sometimes semantic) specification of a language.

Language workbenches With a wealth of language construction tools, a need arose for comprehensive tools that integrated these different solutions and guided the development of languages. Fowler described the trend of integrating the development and use of DSLs into a single IDE environment, and introduced the term *language workbenches* for these tools [16]. In [18] he described this development as follows:

> "Whereas external and internal DSLs have been around for longer than I've been programming, language workbenches are a much newer animal. These tools support DSL creation not just in terms of parsing and code generation but also in providing a better editing experience for DSL users."

Fowler studied a number of practical, modern examples of language workbenches that allow developers to define and use text-based DSLs, including the Meta Programming System (MPS)[15, 25] and Intentional Programming [43]. In his article he also spoke of visual editor environments such as MetaEdit+ [33] and DSL Tools [9], but as these have a very different programming model and do not support text-based languages, we will not discuss them here. Fowler described that language workbenches greatly increase the cost-effectiveness of developing a new language, perhaps even to the point that they can be developed for a single application as sometimes strived for in language-oriented programming [16, 54]. Rather than using a pure text representation, the workbenches Fowler described store the abstract representation of a DSL program, and use syntax-directed (or projectional) editing to manipulate this representation directly. Based on an abstract representation of a program, these workbenches can analyze a DSL program, perform transformations on it, and may show different views.

While it is very important to maintain an abstract representation of a program to enable IDE features such as those made popular by IntelliJ, this does not imply that it should be the principal *storage* representation of programs, certainly given the disadvantages of that approach. Fowler noted the need to be able to store incomplete and contradictory information in the abstract representation, which is not trivial in this model. Other disadvantages include the lack of free text editing; incompatibility with standard, text-based version control systems and issue trackers; and having no way to import artifacts from other (possibly legacy) tools or to edit programs with other tools (leading to vendor lock-in). A free text editing approach, based on modern parser generators, seems much more attractive, since it avoids these problems without precluding the advantages of a language workbench.

Requirements For a language workbench based on freely editable, textual languages, we identify the following requirements:

(1) It must provide an integrated environment for both defining languages and using generated editors.
(2) Conversely, it must be possible to deploy generated editors separately from the workbench for use by "end developers," who may not be interested to work in a meta-programming environment.
(3) The environment must provide state-of-the-art IDE facilities. It should provide a substantial number of modern, language-specific editor services such as automatic indentation and bracket insertion, on-the-fly error markers, reference resolving, and content completion. Many of these services require an abstract representation of a DSL program; the editor should schedule parsing and semantic analysis in the background.
(4) The environment should support efficient, agile language definition, through incremental and selective development of IDE services, which requires separation of concerns between language specifications and pure IDE logic.

Related work The Meta-Environment [34, 44] was one of the first tools that could be described as a language workbench (avant la lettre), combining language specification using ASF+SDF [12] and generated editors for using these languages (1). While it supported the construction of editors for "end developers," these could never really escape the meta-environment (2). Conceived in the early nineties, it did not yet support modern IDE features (3) based on real-time parsing and semantic analysis as programs are edited; error recovery was unavailable for the generated GLR parser at the time. Rather, it required developers to save a file and wait for a list of errors in a separate view. While it allowed the ASF+SDF language to specify language syntax and semantics, the meta-environment offered little opportunity to customize generated editor services (4).

More recent endeavors, which fuse language specification and the construction of modern, interactive IDE components (3), are EMFText [22], MontiCore [36] TEF [55], and Xtext [14].

These approaches all follow the same general architecture. They define their own language for the description of grammars. They may allow annotations in the grammar for the description of syntactic editor services [36], or, by annotating lexical use-def relations, basic semantic editor services [14, 22]. From this grammar they generate a new, separate Eclipse plugin project (2). However, rather than providing a truly integrated environment, they require a second Eclipse instance to load this plugin (1). For IDE support beyond the basic services that can be derived from the grammar, the workbenches allow developers to write fragments of Java code to customize the generated plugin (4). The workbenches either use generated Java classes or an Eclipse Modeling Framework (EMF) metamodel [6] for the abstract syntax. Transformations are carried out using Java visitors or external, EMF-based tooling (4). The tools include string template engines for code generation.

Spoofax In this paper we present Spoofax, a language workbench that enables efficient, agile development of software languages with state-of-the-art IDE support based on concise, declarative specifications.

Spoofax is an integrated environment for the specification of languages and accompanying IDE support in Eclipse. Generated editors can be *dynamically loaded* into the 'meta' Eclipse instance enabling smooth switching between development *of* the language and development *with* the language under construction. Spoofax also supports the generation of a stand-alone plugin for the language under construction that can be deployed to "end developers" without exposing the meta-programming facilities.

Spoofax supports a wide range of editor services based on tightly integrated, real-time application of syntactic and semantic analyses. Analyses are based on the structured abstract representation provided by a live parse of the text in the editor, which uses a parser scheduled in a background thread. Error recovery [11, 29] ensures that editor services function even in the presence of (multiple) syntactic errors. Origin tracking [46] techniques are used to relate the results of analysis back to the text in the editor without requiring preservation of layout information in the specification of analyses and transformations. These and other techniques for the implementation of editor services have been factored into language-parametric components, allowing language developers to focus purely on the language-specific parts of a compiler and IDE.

Spoofax supports language definition *with* declarative domain-specific languages. The modular, declarative syntax definition formalism SDF [21, 49] is closed under composition, ensuring support for language extensions and embeddings [5]. The Stratego transformation language provides a unified formalism for *concise* specification of analysis, transformation, and code generation, enabling reuse of analysis rules for multiple purposes, including dynamic rules [3] for context-sensitive analysis and transformation. We have developed idioms for language specification based on rewrite rules that can be used in batch compilation as well as in interactive editor services. Editor descriptor DSLs provide the bridge between specification of syntax and semantics and the language parametric editor service components, providing a pluggable interface supporting the language engineer in adding new operations to the editor.

To show that our approach is practical, we describe the specification of a web language and report on practical experience with the implementation of other languages and integration with external tools.

The Spoofax language workbench is available from `http://spoofax.org`.

Previous work The Spoofax project started in 2007 with the development of Eclipse editors dedicated to Stratego and SDF [27]. In order to provide IDE support for languages built *with* Stratego and SDF, we developed a prototype of a new Spoofax environment built from scratch, described in [30], where we showed how DSLs can be used to define presentational editor services, and how such definitions can be derived from a grammar. We also sketched an interface for error markers and reference resolving. The present paper shows how a single semantic description based on rewrite rules can be used for both compilation and interactive editor services such as error markers, reference resolving, and content completion. The new Spoofax environment comes with full-featured, "bootstrapped" IDE support for the meta-languages used for language specification, as well as meta-programming features such as the ability to apply transformations directly from the environment.

Outline We proceed as follows. We first describe the architecture of Spoofax and the general anatomy of Spoofax language definitions in Section 2. In Section 3 we discuss syntax definition and the specification of syntactic editor services. In Section 4 we discuss the definition of language semantics: analysis, transformations, code generation, and editor services based on these techniques. We report on experience with language development using Spoofax in Section 5. In Section 6 we elaborate on the implementation of the language workbench. Finally, we discuss related work and directions for future work in Sections 7 and 8, and conclude in Section 9.

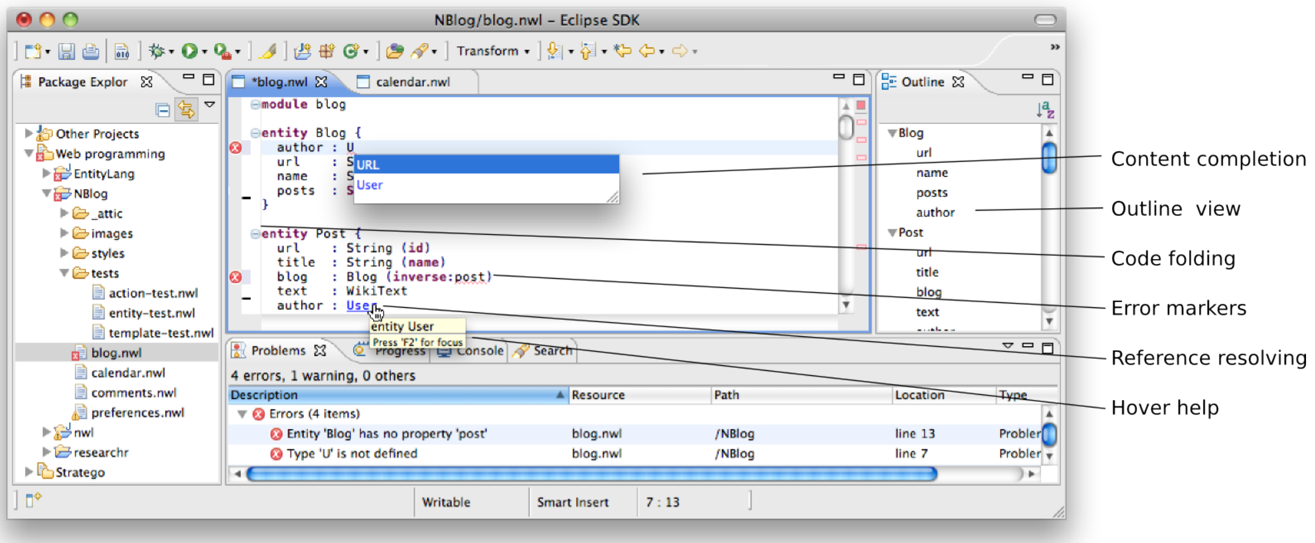
2. An Overview of Spoofax

In this section we give an overview of Spoofax from the point of view of three categories of software developers. "End developers" of a Spoofax IDE work with the editor services specialized to the their (domain-specific) language. The developers of Spoofax itself maintain its architecture and language-parametric components. Language engineers use Spoofax to develop a language definition, i.e. the language-specific elements of an IDE.

2.1 Editor Services

Modern IDEs provide a wide variety of language-specific editor services, which are based on tightly integrated, real-time application of syntactic and semantic analysis. Figure 1 shows a selection.

The editor checks the syntax of the program text, marks syntactic errors inline, and highlights text elements based on the syntactic structure *as the developer types*. The syntactic state of the parser at the cursor is used for editor services such as syntax completion, automatic bracket insertion, bracket highlighting, automatic indentation, and comment insertion. The abstract representation provided by the parser enables code folding, the outline view, and navigation using the quick outline feature.



```
module blog

entity Blog {
  author : U
  url    : S
  name   : S
  posts  : S
}

entity Post {
  url    : String (id)
  title  : String (name)
  blog   : Blog (inverse:post)
  text   : WikiText
  author : User
```

URL
User

entity User
Press 'F2' for focus

```
4 errors, 1 warning, 0 others
Description                             Resource   Path     Location  Type
Errors (4 items)
  Entity 'Blog' has no property 'post'  blog.nwl   /NBlog   line 13   Probler
  Type 'U' is not defined               blog.nwl   /NBlog   line 7    Probler
```

Figure 1. Editor services for a web language.

Based on live semantic analysis of the abstract representation produced by the parser, the editor displays error and warning markers in the code. Program navigation and understanding is supported by reference resolving, occurrence highlighting, and hover help, which use semantic analysis to reveal relations between elements of a program. Content completion shows the developers the valid ways to complete the current construct. Transformation and code generation, using the results of semantic analysis, can be triggered each time the editor is saved, or on demand through the "Transform" drop down menu or using context menus.

In addition to the language-specific editor services provided by a Spoofax plugin, Eclipse is an extensible environment that offers many language-generic development facilities such as plugins for version control, build management, and issue tracking, and the package explorer view (left of Figure 1) that gives an overview of all projects and is used for resource management.

2.2 Component Architecture

Traditionally software languages are developed first as stand-alone compilers and IDEs are later added, typically requiring a significant reimplementation of many of the ingredients of the compiler to realize the implementation of editor services. The components of a compiler — parser, semantic analysis, transformations, and code generation — also play a central role in editor services based on the abstract syntax and semantic analysis of a program. Spoofax has been designed to factor out language independent implementation knowledge into the generic Spoofax libraries. Furthermore, language-specific definitions are defined such that they can be reused in several IDE components. Figure 2 gives an overview of basic compiler components (marked with an asterisk) and editor services in an IDE. The dependencies between these components can be characterized as *generative dependencies*—a component can be automatically derived from another— and *usage dependencies*—a component calls another.

The grammar and parser are at the root of the dependency graph in Figure 2 (A), since the syntactic structure of programs is the basis for the implementation of all other services. In particular, the services for presentation in Figure 2 (B) and editing in Figure 2 (C) are automatically derived from the grammar. These services can then be customized, or re-written from scratch, as desired. Key for the derivation of functionality from a grammar is the use of a declarative syntax formalism. Semantic actions or escapes to external functions, which are sometimes used with parser generators, make it hard to reason about the structure of a grammar for other purposes. In the implementation of Spoofax we use SDF [21, 49]. Another essential component for an editor is error recovery, to ensure that editor services based on the structure of the program keep working in the presence of syntax errors. In [11, 29] we showed how a *permissive grammar*, a grammar with error recovery rules, can be derived from a declarative SDF grammar, ensuring good error recovery even for complex grammars composed of multiple embedded languages or extensions. In Section 3 we describe how SDF grammars are defined and how customizable editor services can be derived from them.

The semantic services cannot be derived from the grammar, since they depend on an interpretation of the syntactic structure of programs. *Name analysis* is a central component, which is reused in all other semantic editor services. Name analysis resolves the declaration of names in a program according to the scope rules of the language.

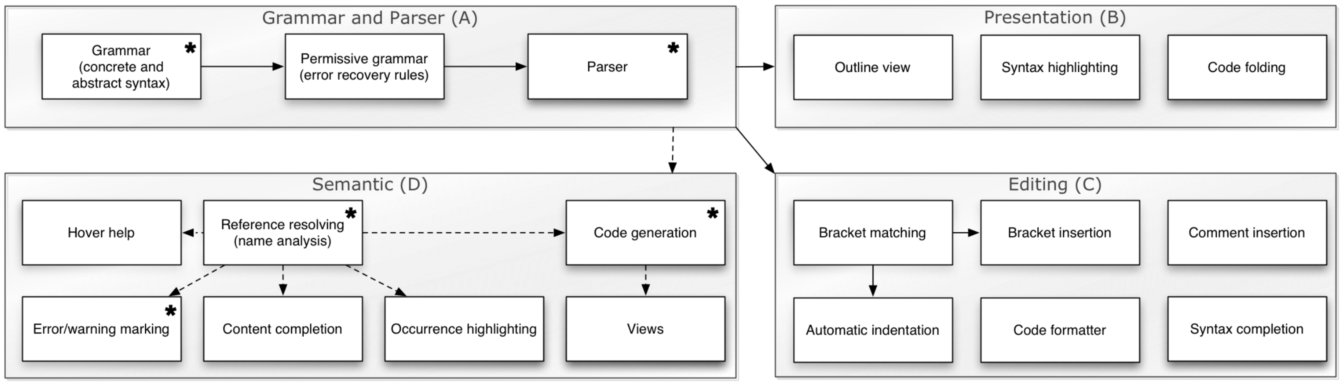


Figure 2. Relations between IDE components. Dependency flow is indicated with arrows; generative dependencies are indicated with a solid line. Components with an asterisk are generally also part of traditional batch compiler implementations.

Custom	Generated
Syntax definition	
Lang.sdf	Common.sdf
Editor service descriptors	
Lang.main.esv	
Lang-Builders.esv	Lang-Builders.generated.esv
Lang-Colorer.esv	Lang-Colorer.generated.esv
Lang-Completions.esv	Lang-Completions.gen...esv
Lang-Folding.esv	Lang-Folding.generated.esv
Lang-Outliner.esv	Lang-Outliner.generated.esv
Lang-References.esv	Lang-References.gen....esv
Lang-Syntax.esv	Lang-Syntax.generated.esv
Semantic definition	
lang.str	
check.str	
generate.str	

Figure 3. Language definition components.

2.3 Structure of a Language Definition

A Spoofax language definition is an Eclipse project that defines the language-specific elements of an IDE, reusing the language-parametric components from the Spoofax infrastructure.

Figure 3 gives an overview of the default structure of a language definition project. Each of the three main components – syntax, service descriptors, and semantics – is defined in a number of modules. Developers are free to organize these how they wish, but the default layout separates the different concerns into different files, allowing developers to quickly familiarize themselves with the components and interfaces of a language definition.

An important design principle in the combination of derived and handwritten files has been to clearly indicate in the file name which files are generated. These files are regenerated every time the project is rebuilt and should not be edited by the language developer. To ignore specific rules in the generated file, they can be disabled or redefined in the accompanying handwritten file. If the generated file is not used at all, it can simply be removed from the list of imported descriptor files.

The *syntax* is defined using SDF [21, 49]. The default project comes with a skeletal language with four production rules (shown in Figure 4), and a module `Common.sdf` with default rules for comments and lexical patterns such as strings and identifiers.

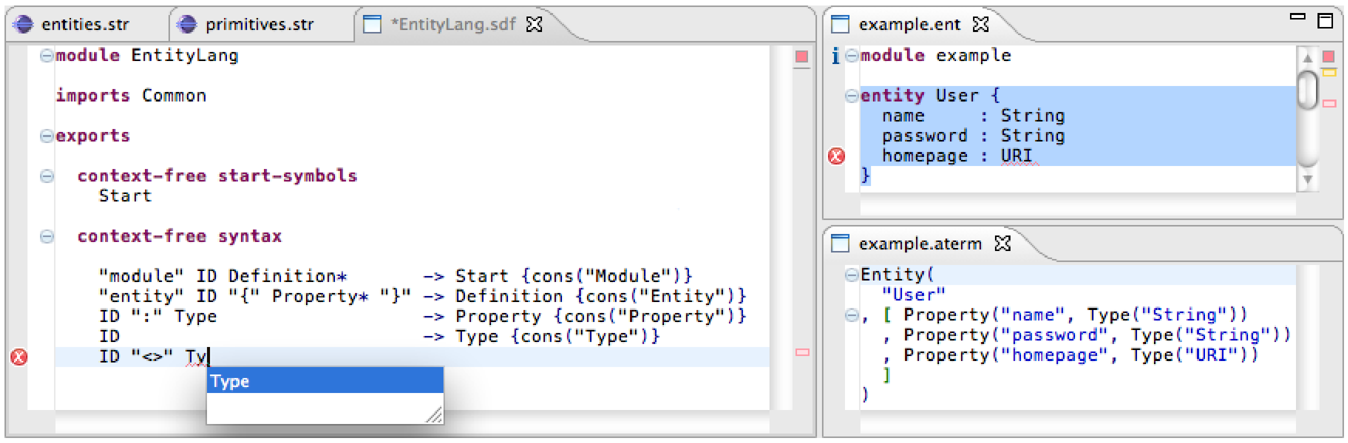
Editor services are defined using declarative, rule-based editor descriptor languages. These can be used to define presentation or editing services, and can describe the interface of semantic editor services (describing what transformations to use for which service, and which views can be shown for a language). Derived services are maintained in separate `.generated.esv` files, and provide basic functionality (or, at the very least, examples) for these services based on the grammar. Not all services can be derived, but these files are also a source of documentation and examples.

Semantic definitions are specified using Stratego [3], which provides an integrated solution for analysis, transformation, and code generation rules. Spoofax separates editor service specifications and the transformations that implement them. Editor service descriptors specify *which* transformations to apply, while the Stratego specifications specify *what* these should do. This design ensures flexibility in the implementation of services and allows for possible future integration with other meta-programming languages and frameworks.

We discuss the three categories of definitions and their relations in more detail in the following sections.

2.4 Agile Language Development

The architecture of the Eclipse platform is based on the OSGi component model, in which each plugin is (usually) a JAR containing Java classes, a plugin manifest, optional descriptor files, and auxiliary resources, such as images. The descriptors specify which parts of the Eclipse framework a given plugin extends, and which parts of the plugin may be extended by other plugins. The OSGi model implies distributing plugins as static JARs. The normal workflow cy-



```
module EntityLang

imports Common

exports

  context-free start-symbols
    Start

  context-free syntax

    "module" ID Definition*         -> Start {cons("Module")}
    "entity" ID "{" Property* "}"  -> Definition {cons("Entity")}
    ID ":" Type                     -> Property {cons("Property")}
    ID                              -> Type {cons("Type")}
    ID "<>" Ty
              Type
```

```
module example

entity User {
  name     : String
  password : String
  homepage : URI
}
```

```
Entity(
  "User"
, [ Property("name", Type("String"))
  , Property("password", Type("String"))
  , Property("homepage", Type("URI"))
  ]
)
```

Figure 4. Multiple editors, side by side, in the same Eclipse IDE instance: the definition of an entity language (left), an editor for the entity language itself (upper right), and the abstract syntax of the selected entity (lower right).

cle for plugin developers is to declare new extensions in the `plugin.xml` descriptor file, implement these in Java, and test the plugin in a second instance of Eclipse, which is detrimental to a rapid development process.

One IDE instance Language definitions in Spoofax are based on the Eclipse plugin project model: each language definition includes a plugin manifest and descriptor files that allow it to be distributed to "end developers" as a normal Eclipse plugin. However, to enable agile language development we use a very different workflow model than that of standard Eclipse plugin development. By using language-parametric editor services that dynamically load and update language-specific service specifications (described in more detail in Section 6), we can use generated editors for a language in the same environment (Eclipse instance) in which we edit the language definition itself. Figure 4 illustrates how a grammar (left) and a generated editor (upper right) can be used side by side. The editor is fully functional and includes semantic services also defined in the same environment. The lower right editor illustrates an abstract syntax view for the selection in the generated editor that is updated in real-time as the selection is edited.[1] The same view can be used to inspect (intermediate) results of transformations.

Inductive design Rather than designing a complete DSL "on paper," before its implementation, it is good practice to incrementally introduce new features and abstractions through a process of evolutionary, inductive design [16, 51]. In the context of a language workbench, this means that DSL programs and the DSL itself evolve together. This enables quick turn-around time for the development of the DSL and the subsequent gradual extension as new applications are developed, and new insights into the domain are acquired.

[1] While a graphical abstract syntax view can be visually appealing, we opt for an automatically formatted textual view instead, as it is much more concise, conveys the same information, and benefits from standard, textual editor services. Moreover, the same textual representation is also used in (and can be copy-pasted to) specifications of analyses and transformations.

The Spoofax environment assists in the initial creation of a new language using a wizard that simply takes the name of the language, the file extension it uses, and a package name. The wizard then creates a new Eclipse project with a skeletal language definition. From this point, new language features can be added through an iterative development process. New language constructs can be added to the grammar. These features can be directly used in the editor for the language.

A new language project created by the wizard includes standard Eclipse plugin configuration files (these are typically not changed by language developers), as well as specification files native to the Spoofax environment. Developers can then define editor services and define semantics for these features. Some editor services are automatically derived from the grammar; their specification can be adapted as desired.

An important aspect of the Spoofax architecture is that it allows for selective development of editor services. Developers can freely select what services to implement: the editor can also be used with a subset of all features. For example, developers may forgo sophisticated semantic analyses and transformations, and simply define code generation by a very direct mapping of abstract syntax to target code using string templates. Reuse is key for efficient language development, which means there are some dependencies between services (as seen in Figure 2), but most can be completed individually, allowing the language and IDE to be evaluated and used at every stage of development.

Language understanding with views The specifications of most editor services, in particular those for semantic services based on analyses and transformations, are defined in terms of a textual abstract representation of programs. Using the abstract syntax view (Figure 4), developers can inspect the abstract syntax of a text selection or file.

The abstract syntax view of Figure 4 is not a built-in Spoofax feature, but it is a view that is defined with the default, skeletal language definition. Views show the results of transformations (as indicated by the relation in Figure 2). When a view is opened, it is automatically placed to the side of its source file, allowing developers to view both at a glance. Views are implemented using standard, textual editors (either for DSLs or languages such as Java that live in the Eclipse environment). They show either the abstract syntax of a transformation or the concrete syntax (e.g., standard Java code). The default view for showing the abstract syntax is defined by showing the result of the "identity" transformation, i.e., code is only parsed, not transformed.

Views are an important aspect of our architecture and a requirement for agile language development: they are essential for awareness of the abstract representation of a language, and can be used to show (intermediate) results of analyses and transformations independent of other editor services.

2.5 Example Domain-Specific Language

In the following sections we use the NWL language[2], a subset of WebDSL [51], to illustrate key points of language definition. NWL covers several aspects of web programming i.e. entity declarations (data modeling), properties with inverse relations, parameters and variables, expressions, template definitions, page navigation, and several types of template elements. However, in this paper we focus only on definitions of entities and actions (which are analogous to data type definitions and functions in other languages).

3. Syntax

The central implementation artifact for any textual language is the parser, which can be generated from a grammar.

In Spoofax, the grammar has the following roles:

1. it specifies the concrete syntax (keywords etc.)
2. it specifies the abstract syntax (the data structure used for analysis and transformation of programs written in the language)
3. it is used to derive editor services for presentation and editing that can be customized by the developer

We use SDF [21, 49] to define grammars. SDF grammars are declarative, highly modular, combine lexical and context-free syntax into one formalism, and can define concrete and abstract syntax together in production rules [32].

SDF productions take the form $p_1 \ldots p_n$ `->` s and specify that a sequence of strings matching symbols p_1 to p_n matches the symbol s. Productions can be annotated with a constructor name n to uniquely identify them in the abstract syntax using the `{cons(`n`)}` annotation. Other annotations include `{left}` and `{right}` to specify the associativity of operators, and `{deprecated(`e`)}` to mark deprecated syntax with optional explanation e.

[2] The complete definition of NWL is available at `http://strategoxt.org/Spoofax/NWL`.

```
module NWL
imports Common
exports

context-free start-symbols
  Start

context-free syntax
  "module" ID Def*          -> Start {cons("Module")}
  "import" ID               -> Def   {cons("Import")}
  "entity" ID "{" Prop* "}" -> Def   {cons("Entity")}
  "action" ID "(" {Param ","}* ")" "{" Stat* "}"
                            -> Def   {cons("Action")}

  ID ":" Type        -> Param {cons("Param")}
  ID ":" Type        -> Prop  {cons("Property")}
  ID                 -> Type  {cons("SimpleType")}
  "Set" "<" Type ">" -> Type  {cons("SetType")}

  Exp ":=" Exp ";"   -> Stat  {cons("Assign")}
  "for" "(" ID ":" Type ")" "{" Stat* "}"
                     -> Stat  {cons("ForAllEntity")}
  "for" "(" ID ":" Type "in" Exp ")" "{" Stat* "}"
                     -> Stat  {cons("ForAll")}

  "all" "(" Type ")" -> Exp   {cons("ForAllExp")}
  STRING             -> Exp   {cons("StringLit")}
  ID                 -> Exp   {cons("Var")}
  Exp "." ID         -> Exp   {cons("PropAccess")}
```

Figure 5. A grammar for entities and actions in NWL.

Figure 5 shows an abbreviated SDF grammar for the NWL language. The grammar extends the basic entity language of Figure 4 with additional features. NWL modules consist of a module name and a list of `Def` definitions. Definitions can be entity declarations, import declarations, or actions. Actions have a comma-separated list of `Param` parameters and a list of `Stat` statements.

Mapping between abstract and concrete syntax The abstract syntax, used in the specification of editor services, can be represented as first-order terms of the form

```
t ::= "..."           // string literals
  | c(t1,...,tn)      // constructor applications
  | [t1,...,tn]       // lists of terms
```

As an example, consider the first production of the NWL grammar:

```
"module" ID Def* -> Start {cons("Module")}
```

This production has three elements: the literal "module", an identifier name, and a list of definitions. For analyses and transformations we're usually not interested in literals and layout, so only the name and list of definitions are included in the abstract representation:

```
Module("example", [Entity("User", [...])])
```

which corresponds to the abstract syntax of the module at the upper right of Figure 4.

Spoofax generates a parser from the grammar, which produces the abstract representation of a file every time the user presses key and a short delay passes. After the parser completes, all editor services that depend on the abstract representation are updated automatically. Internally, the abstract representation is stored efficiently in memory as Java objects, and maintains full layout and position information for use in services that need it.

3.1 Syntactic Editor Services

Editor services related to presentation and editing can be based directly on the syntax definition (as indicated by the relation in Figure 2). These services can be fully specified using declarative editor service descriptor specifications. Rather than give an exhaustive overview of these descriptors and their features (available online at [1]), we show some examples in this section to give an impression of how a declarative descriptor DSL can concisely describe these services.

Syntax highlighting Default syntax highlighting behavior is derived based on the literals and lexical syntax in the grammar. The colors used for this derived behavior are specified in the generated colorer descriptor, shown in the lower half of Figure 6. It specifies a color for keywords (alphanumeric literals in the grammar), operators (non-alphanumeric literals), strings (lexicals that allow spaces), numbers (lexical numeric patterns), and identifiers (other lexicals). The default colorization works well, but can be customized in the `NWL-Colorer.esv` file. The top half of Figure 6 illustrates custom coloring rules for the `Type` symbol, with specific colors for the `SimpleType` and the `SetType` constructor. Other coloring rules can override the colors for literals and lexicals, and can specify background colors, colors for regions of code rather than single productions, and more.

Code folding and outline view Code folding and the outline view are specified by selecting grammar productions that should be made foldable or shown in the outline view. Figure 7 illustrates some folding rules for the NWL language. Spoofax uses heuristics to automatically derive a generated folding descriptor, based on the logical nesting structure of the language. Currently, productions rules that have an identifier lexical and a list of child elements are included in this descriptor. While not perfect, the heuristic provides a good starting point for a new folding definition. Any undesired definitions in the generated file can be disabled by using the `(disabled)` annotation in the custom specification. The `(folded)` annotation can be used for constructs that should be folded automatically.

Bracket highlighting and insertion By describing pairs of matching brackets and the comment constructs of a language, the bracket highlighting, bracket insertion, and comment insertion features can be enabled for an editor (Figure 8). Bracket pairs are also used to supplement the automatic indentation specification (not shown): the cursor is automatically indented one level if a newline is entered after an opening bracket.

Syntax completion We distinguish syntactic and semantic content completion (the latter is discussed in the next section). Syntactic content completion provides users with completion suggestions based purely on static, syntactic templates. For example

```
completion template:
  "entity " <e> " {\n\t\n}"
```

is a syntactic completion rule for entity definitions. Completion rules are composed of static strings and placeholder expressions. Static strings allow for precise control of the presentation of completions and are enclosed by double quotes. They can use `\n` for newlines or `\t` for one indentation level (following the user's tab/space configuration). Placeholder expressions are indicated by angular brackets. The editor automatically moves the cursor to these expressions once the user selects a completion proposal, allowing the expressions to be filled in as the user continues typing.

```
module NWL-Colorer
imports NWL-Colorer.generated
colorer
  Type.SimpleType : cyan
  Type.SetType    : gray

module NWL-Colorer.generated
colorer
  keyword    : magenta bold
  identifier : default
  string     : blue
  ...
```

Figure 6. Syntax highlighting rules for NWL.

```
module NWL-Folding
imports NWL-Folding.generated
folding
  Start.Module
  Definition.Entity
  Definition.Action
```

Figure 7. Folding rules for NWL.

```
module NWL-Syntax
language
  line comment  : "//"
  block comment : "/*" * "*/"
  fences        : ( ) { }
```

Figure 8. Comment and bracket definition rules for NWL.

4. Analysis and Transformation

Semantic analysis has two key roles in the implementation of programming languages. First, the analysis checks if programs program are (type) consistent, reporting errors if they are not. Second, it provides semantic information for use by compilers, IDEs, and other language-specific tools.

In IDEs, semantic analysis forms the basis for all semantic editor services. There are two forms of semantic analysis that are particularly important for IDEs: name analysis and type analysis. Name analysis binds each identifier occurrence to its declaration. Name analysis is exposed directly in the IDE in the form of reference resolving: press and hold Control and hover the mouse cursor over an identifier to reveal a blue hyperlink that leads to its declaration. Type analysis determines the type of expressions and is important for reporting errors and for context-dependent code generation.

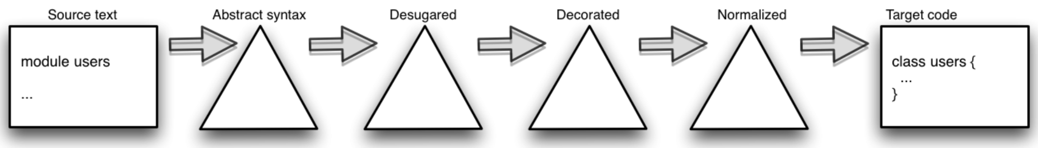


Figure 9. Staged compilation: source code is parsed, transformed, and finally printed to target code.

Other analyses such as flow and pointer analysis may also have a role in marking errors and warnings or for optimization of the generated code, but in this paper we focus on name and type analysis because of their central and crucial role in both compilation of languages and for editor services.

Like many traditional compilers, we employ a staged architecture of analyses and transformations, as illustrated in Figure 9. First programs are parsed, then syntactic sugar is eliminated, and then they are analyzed, creating abstract syntax trees decorated with semantic information. Semantic editor services such as reference resolving and error marking operate on these decorated trees. After the analysis, the tree can be further normalized to a core form, and finally code generation rules can generate resulting code.

In the remainder of this section, we first introduce the Stratego transformation language, and then show idioms for using Stratego rewrite rules to concisely and declaratively specify analyses and transformations for use by editor services and for code generation.

4.1 Stratego

We use the Stratego program transformation language [3] to describe the semantics of a language. The Stratego language is based on the paradigm of term rewriting with programmable rewriting strategies introduced by [52]. Basic transformations are defined by means of conditional *term rewrite rules* of the form

```
r : t1 -> t2 where s
```

with `r` the name of the rule, `t1` and `t2` first-order terms, and `s` a *strategy expression*. A rule applies to a term when its left-hand side `t1` matches the term, and the condition `s` succeeds, resulting in the instantiation of the right-hand side pattern `t2`. Otherwise the application *fails*. Unconditional rules have no `where` clause, others may have multiple that must all be satisfied.

In addition to checking applicability constraints, the `where` clause of a rule can perform computations that may be used in the right-hand side of the rule. For example, in the rule schema

```
r : t1 -> t2 where t3 := <s> t4
```

the term `t4` is transformed by application of a strategy or rule `s`, matching against and binding variables in the pattern `t3`. Term `t4` may use variables from the left-hand side `t1`, and right-hand side `t2` may use `t3`.

Rewrite rules can concisely express small transformations based on the abstract representation of a program. Using content completion of terms, based on the syntax of a language, and by providing views of the abstract syntax and the results of transformations, the Spoofax environment assists in writing these rules.

More complex transformations can be created by composing rules using *strategies*. Many strategies can be compared to visitors in object-oriented programming in that they guide the application of rules in a tree. A strategy is essentially a partial function from terms to terms. If a strategy is not defined on a term it is said to *fail*. Failure arises from the failure of rewrite rules to apply to terms. Strategies are composed from basic combinators such as the identity transformation `id`, sequential composition `s1; s2` and deterministic choice `s1 <+ s2`. The Stratego standard library provides a number of strategies for general use, such as `map(s)` that applies a strategy expression `s` to each element of a list, and `alltd(s)` that descends down the branches of a term up to the points where `s` can be successfully applied, returning the complete term after transformation.

Context-sensitive transformations can be expressed by means of *dynamic* rewrite rules [4], which are instantiated at run-time, as illustrated by the following schema:

```
r : t1 -> t2
where rules(dr : t3 -> t4)
```

The dynamic rule `dr` is defined when `r` is applied to a term matching `t1`. Any variables that `t3` and `t4` share with `t1` are then inherited by the instantiation of `dr` (concrete examples follow below).

4.2 Desugaring

Desugaring rules can simplify the abstract representation, transforming higher-level constructs to more general, lower-level constructs, or mapping constructs with multiple forms to a single, canonical form. They can also be used to migrate deprecated language constructs to newer alternatives. This way, later analysis and transformation stages only need to focus on a subset of all language constructs.

The full WebDSL language supports various different iteration expressions and statements with optional filters and ordering clauses [51]. In our NWL subset we provide three basic iteration constructs without filtering or ordering, shown in the grammar of Figure 5. The `ForAllExp` expression and the `ForAllEntity` statement iterate over all instances of a given entity type. The `ForAll` statement is more general and iterates over a given expression. To avoid having to write typing, transformation, and code generation rules for each of these variations, we use desugaring rules to transform similar constructs to the most general form.

```
desugar-top = innermost(desugar)

desugar:
  ForAllEntity(x, t, s*) ->
  ForAll(x, t, ForAllExp(t), s*)

desugar:
  Call(x) -> CallArgs(x, [])
```

Figure 10. Simple desugaring rules.

Figure 10 shows a `desugar` rule that transforms the `ForAllEntity` statement to the more general form. Another definition of the rule transforms action calls without arguments to action calls with an empty list of arguments (omitted in the syntax definition). More sophisticated, context-sensitive transformations, such as type inference for the `ForAll` loop, can be applied after semantic analysis of the program. The `desugar` rules are exhaustively applied in an innermost fashion by the `desugar-top` strategy.

4.3 Reporting Errors and Warnings

There are a number of concerns in checking a program for errors and reporting them to the user:

– *Context*: identifying points in the code to check
– *Assumptions*: only report an error if certain assumptions hold (validating the context and avoiding spurious errors)
– *Constraints*: checking for constraints at the context
– Formulating an *error message*
– *Attribution* of the error to a particular character range in the source text (usually, only part of the context is marked, such as the name of an erroneous method)

In Spoofax, error checking is a transformational process: it transforms an abstract syntax tree to a list of errors (also a tree, containing messages and attributed tree nodes). We use regular Stratego rewrite rules to transform parts of the tree to errors. These *check rules* encapsulate all of the above concerns, and adhere to the following idiom:

```
check:
  context -> (target, $[error message])
  where assumption
  where require(constraint)
```

The rule checks whether for the given `context` (the term under scrutiny) the `constraint` is satisfied, given that the `assumption` holds. The `require(s)` operator is sugar defined as follows:

```
require(s) = not(s)
```

That is, the check rule *succeeds* (producing the specified `error message`), if the `assumption` succeeds but the `constraint` *fails*.

The right-hand side is a pair consisting of target term and an error message. The `target` is (typically) a subterm of the `context` and denotes the term to which the error message should be attributed. The error message should explain that the `constraint` does not hold. The error message is a string typically composed using *string interpolation*, i.e. `$[...]` is a string consisting of all literal characters between the quotes, except for escapes between `[...]`. For example, if `t` is bound to `"Blog"` and `p` to `"size"` then

```
$[[t] has no property [p]]
```

evaluates to the string `"Blog has no property size"`.

```
check-top = collect-all(check)

check: Var(x) -> (x, $[Variable [x] is not declared])
  where require(type-of)

check: SimpleType(x) -> (x, $[Undefined type [x]])
  where require(is-simple-type)

check: PropAccess(e2, p) -> (p, $[[t] has no property [p]])
  where t := <type-of> e2
  where require(type-of)

check: ForAllExp(t) ->
       (t, $[For loop requires entity type argument])
  where <is-simple-type> t
  where require(<is-entity-type> t)

check: SetType(t) ->
       (t, $[Set requires entity type argument])
  where <is-simple-type> t
  where require(<is-entity-type> t)
```

Figure 11. Consistency checking rules.

Figure 11 gives a selection of check rules for the NWL language. The `check-top` strategy controls the application of these rules: it collects a list of all tuples resulting from successful applications of the check rules. For the specification of assumptions and constraints, these rules use a number of helper rules that are defined by the name and type analysis (which we describe in the next subsection). The `type-of` helper is a rule that returns the type of an expression. If the type cannot be resolved (indicating an error in the program), `type-of` *fails*. For example, the first check rule of Figure 11 requires that `type-of` succeeds for the context. If it does not, an error is reported. Another helper is `is-simple-type`, used in the second check rule: it only succeeds if the given type is a declared entity type (as opposed to a set type). This helper is also used to distinguish the case where constructors are undefined or just non-entity types.

4.4 Binding Transformations to Editor Services

Transformation with origin information For the check rules it is important to maintain the relation between the original source location and the term an error is attributed to. Figure 12 illustrates the different steps of the error checking strategy. First, the source code is parsed, desugared, and analyzed, producing the abstract syntax tree to the left. Then the `check-top` strategy is applied, producing the tree to the right, with pairs for all attributed terms and error messages. Throughout the desugaring, analysis, and error checking process, the relations between the original source positions and the terms are maintained, as shown by the dashed lines. Using these relations, all errors can be reported in the source code locations that correspond to the attributed terms.

The origin relations are maintained automatically by the Spoofax environment, allowing language developers to write concise, position information-agnostic transformations

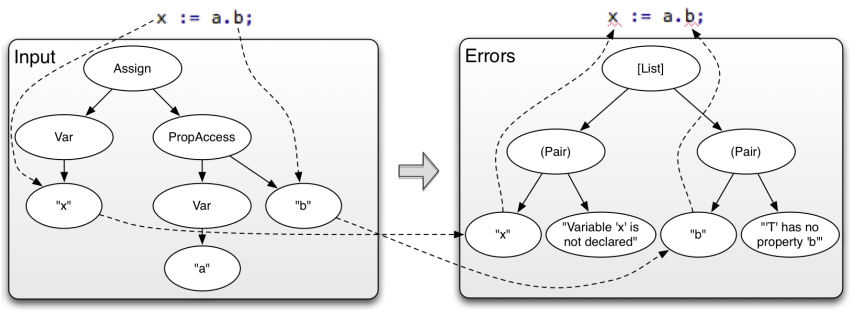


Figure 12. Transformational error checking with origin information.

```
editor-analyze: (ast, path, project-path) ->
                (ast'', errors, warnings, notes)
  where editor-init;
        ast'     := <desugar-top> ast;
        ast''    := <declare-top> ast';
        errors   := <check-top> ast'';
        warnings := <check-warning-top> ast'';
        notes    := <check-note-top> ast''
```

Figure 13. Control rule for semantic analysis.

rather than keeping track of the original position manually. Position information is implicitly passed along with transformations through a form of *origin tracking* [46]. Origin tracking is a general technique that can be applied in term rewriting systems to implicitly maintain a link to the originating term after it has been rewritten to a new term.

Origin tracking as introduced by Van Deursen et al [46] has originally been used to trace back errors to their original source, much like we do in our error checking transformation. However, as we show in the remainder of this section, the technique can also be used to for high-level specifications of analyses and transformations used by editor services.

Control rules The transformation of an abstract syntax tree to a list of errors is controlled by the analysis *control rule*. Control rules are regular Stratego rewrite rules with a fixed signature that form the interface between the IDE and the Stratego transformation specification.

The control rule for semantic analysis and error reporting is shown in Figure 13. The rule is given information about the file that is being analyzed, in the form of a tuple with the abstract syntax tree of the file to be checked, its project-relative path, and the file system path of the project itself. The rule transforms this input to a new abstract syntax tree, decorated with semantic information, and lists of errors, warnings, and informational notes to be marked in the program. In the `where` clause, the rule first resets the current analysis environment using the built-in `editor-init` strategy. Corresponding to Figure 9, the rule then desugars the input abstract syntax tree and analyzes it, by calling the `declare-top` strategy (given below). It then collects all markers for the resulting (decorated) abstract syntax tree using the check strategies.

Semantic editor descriptors Editor descriptor files determine which control rule is used for which service. For example, the `NWL-Builders.esv` file specifies:

```
observer: editor-analyze
```

which means that `editor-analyze` is used as the `observer` control rule for the semantic analysis and error reporting. Control rules can also be specified for when files are saved (`on save`), for when a view is opened (`builder`), for reference resolving (`reference`), and for each of the other semantic services of Figure 2.

4.5 Name and Type Analysis

Name analysis binds each identifier occurrence to its declaration. Based on name analysis, type analysis identifies the types of expressions. Name analysis is also used for semantic editor services and code generation, as illustrated in Figure 2. Together, name and type analysis form the basis for consistency checking rules as those seen in Section 4.3.

As name analysis and type analysis are generally used together, it is common practice for compilers to combine the two analyses. Rather than locating the definition site for identifier occurrence, the name analysis then directly collects a mapping of names to types, and the type analysis determines the types of surrounding expressions.

In Spoofax we use the name analysis independently from the type analysis in semantic editor services. For this reason it is important to separate it from the type analysis. In this section we present an idiom for separate specification of name and type analysis using rewrite rules and dynamic rules. For simplicity, we divide the name analysis in two stages: first we collect all globally visible names, and then we analyze scoped names.

Unscoped name analysis Globally visible names can be stored directly as dynamic rules that map an identifier to its declaration site. For NWL, we use the `EntityDef` and `ActionDef` dynamic rules for global names (Figure 14). The `declare-top` strategy controls the stages of the name analysis. It first declares all primitive types, then applies the `declare` rule to the outermost definitions, and finally calls `rename-top` to analyze scoped names (shown below).

```
declare-top =
  declare-primitive-types;
  alltd(declare);
  rename-top

declare-primitive-types =
  rules(
    PrimitiveDef: "String" -> "String"
    PrimitiveDef: "Int"    -> "Int"
    ...
  )

declare: d@Entity(x, p*) -> d
  where rules(EntityDef : x -> d)

declare: d@Action(x, param*, stat*) -> d
  where rules(ActionDef : x -> d)

declare: Import(x) -> Import(x)
  where open-import(
          resolve-import
        , parse-file
        , declare-top
        )

resolve-import: Import(x) -> $[[<project-dir>]/[x].nwl]
```

Figure 14. Name analysis for globally visible names.

The declare rules match entities and actions, and define dynamic rules that map each name x to its definition site d.

For import declarations, Spoofax provides a generic open-import strategy (used in Figure 14, bottom). This strategy ensures that files are cached after they are parsed and avoids cycles in the dependency graph. The open-import strategy takes three strategy expressions as its arguments, that respectively resolve the filename of an import (resolve-import), parse the file (parse-file), and analyze the file (declare-top).

Scoped name analysis For scoped names, we base our analysis on the notion of consistent renaming [53], which is the task of renaming all names in a program such that they are unequal to all other names that do not correspond to the same declaration site. However, rather than directly changing the names in the tree, we add annotations that satisfy this uniqueness requirement. This way, the abstract syntax tree remains the same modulo annotations, aiding in debugging and traceability of analysis and code generation. As an example, Figure 15 shows how an action definition in concrete syntax (top), abstract syntax (middle), and as abstract syntax with renaming annotations (bottom). After renaming, each local name combined with its annotation is globally unique: for instance, we can distinguish "s"{"a_0"} from any other identifier s defined elsewhere.

Figure 16 shows *renaming rules* that add consistent naming annotations to local variables. The first definition of rename (Figure 16 (A)) operates on action parameters. It replaces the name x of a parameter with the annotated name x{<new>}, where <new> generates a fresh, globally unique name. The VarDef dynamic rule is defined to map the annotated name to the definition site. The RenameId dynamic rule records the renaming for the current scope: for each following occurrence of x, it should given the name y (Figure 16 (B)). Any names that are not declared will not be renamed, and are reported as errors in the check stage as VarDef is never defined for them.

```
action add(s : Set<User>) {
  for (u : User in s) { ... }
}
```

```
Action("add",
  [Param("s", SetType(SimpleType("User")))],
  [ForAll("u", SimpleType("User"), Var("s"), ...)]
)
```

```
Action("add",
  [Param("s"{"a_0"}, SetType(SimpleType("User")))],
  [ForAll("u"{"b_0"}, SimpleType("User"),
          Var("s"{"a_0"}), ...)]
)
```

Figure 15. Consistent renaming of an action definition.

```
rename-top = alltd(rename)

rename: d@Param(x, t) -> Param(y, t)                   (A)
  where y := x{<new>};
        rules(
          VarDef   : y -> d
          RenameId : x -> y
        )

rename: Var(x) -> Var(y)                               (B)
  where y := <RenameId> x

rename: Action(f, p1*, s1*) -> Action(f, p2*, s2*)     (C)
  where {| RenameId:
           p2* := <rename-top> p1*;
           s2* := <rename-top> s1*
        |}

rename: d@ForAll(x, t, e, s1*) -> ForAll(y, t, s2*)
  where {| RenameId:
           y := x{<new>};
           rules(
             VarDef   : y -> d
             RenameId : x -> y
           );
           s2* := <rename-top> s1*
        |}
```

Figure 16. Name analysis using renaming rules.

To respect the scopes of the language, we use *dynamic rule scopes* [4] to reflect scoping construct of the language. This Stratego feature takes the form {| r: ... |} and ensures that any definition of the dynamic rule r made directly or indirectly within the scope is no longer visible after the scope is exited. The rename rule for actions (Figure 16 (C)) uses this feature to scope variables declared within an action. As the RenameId for these variables is undefined at the end of the scope, the variables are no longer visible in any following actions. The last definition of rename combines local renaming and scoping for the for statement.

Type analysis Type analysis can be specified using rules that project a language construct to its type. The type-of rule maps language constructs to their type (Figure 17). Typing rules for literals generally specify a constant type (e.g., for StringLit). Typing rules for declarations can fetch the type from one of the subterms (e.g., for entities or properties). Other typing rules use name analysis rules to fetch a definition and then apply the basic type-of rules (e.g., for

```
type-of: StringLit(x)       -> SimpleType("String")
type-of: Entity(x, prop*)   -> SimpleType(x)
type-of: Property(_, t, _)  -> t
type-of: Param(x, t)        -> t
type-of: ForAll(x, t, _, _) -> t
type-of: ForAllExp(t)       -> SetType(t)
type-of: Var(x)             -> <type-of> (<VarDef> x)
type-of: PropAccess(e, f)   -> t
  where p := <lookup-property(|f)> (<type-of> e);
        t := <type-of> p
```

Figure 17. Typing rules.

```
lookup-property(|f):
  Entity(y, p*) -> <fetch-elem(?Property(f,_,_))> p*

lookup-property(|f):
  SimpleType(y) -> <lookup-property(|f)> (<EntityDef> y)

is-simple-type = is-primitive-type <+ is-entity-type

is-primitive-type: SimpleType(x) -> <PrimitiveDef> x

is-entity-type: SimpleType(x) -> <EntityDef> x
```

Figure 18. Helper rules for error checking.

Var). For compound expressions such as PropAccess, we fetch the property definition corresponding to the expression and return its type. To fetch the property f of a given type we use the lookup-property(|f) helper rule, where the pipe (|) indicates that it receives a *term argument* f (as opposed to a strategy argument).

Other projection rules can test for various properties based on the name analysis. Figure 18 illustrates additional projection rules that are used in the check rules. Of these, the lookup-property rule is of particular interest. It uses the Stratego standard library strategy fetch-elem to try and fetch the property f from an entity definition. A second definition fetches the property given a type instead of an entity.

4.6 Reference Resolving and Occurrence Highlighting

Reference resolving and occurrence highlighting are specified using a control rule that given an identifier, respectively looks up its declaration or all occurrences of that identifier. For reference resolving, this rule is shown in Figure 19. The rule is given the selected node in the tree, its position (specified as a list of child offsets), the decorated abstract syntax tree, and the file system paths. The same tuple is given to the control rules of all other semantic editor services (with the exception of the analysis and error checking control rule). Reference resolving is implemented by simply applying the dynamic rules from the name analysis stage to the selected identifier, using a decl-of helper rule to fetch the appropriate declaration. Occurrence highlighting can be specified in a similar fashion, but is omitted here for reasons of brevity. Both services rely fully on the name analysis specification and only require a small addition to implement the editor service.

4.7 Content Completion

Semantic content completion (sometimes called content assist) provides completion proposals based on the syntactic

```
editor-resolve:
  (selected, pos, ast, path, project-path) -> target
  where target := <decl-of> selected

decl-of: Var(x)        -> <VarDef> x
decl-of: SimpleType(x) -> <EntityDef> x
decl-of: Submit(x)     -> <ActionDef> x
```

Figure 19. Rules for reference resolving.

and semantic context of the expression that is being edited. Content completion can provide suggestions for globally visible names as well as locally scoped names. For this reason, we integrate content completion into the name analysis of Section 4.5.

An important aspect of our design for the content completion service is its interface to the semantic analysis. Content completion suggestions must be provided regardless of the syntactic state of a program: an incomplete expression 'blog.' does not conform to the syntax, but for content completion it must still have an abstract representation. One approach (as taken by IMP [8], for example) is to require the language developer to add special productions to the grammar that map these incomplete expressions to an abstract representation. Unfortunately, this means the developer has to do extra work and has to account for all cases where content completion may be expected in the grammar.

In our approach, we opt for a generic representation of incomplete syntactic expressions: we introduce an artificial COMPLETION term to the abstract representation at the point of the cursor. For example, if a variable is expected at the cursor, a Var(COMPLETION("p")) is placed at the corresponding point in the abstract representation, with p the (possibly empty) identifier prefix that was already typed in. Similarly, compound expressions can take a form like PropAccess("blog",COMPLETION("p")).

In Figure 20 we define content completion for variables and property access expressions by extending the name analysis from Figure 16. We replace the definition of rename-top with one that applies propose-completion before it applies rename. The first definition of propose-completion provides completion suggestions for variables, by taking all *keys* of the RenameId dynamic rule. Stratego provides the derived function all-keys-RenameId to do this. As the rule is evaluated within the context of the analysis, this results in a list of all variables in the current scope, and thus all valid suggestions if a variable is expected at the cursor location. The ContentProposals dynamic rule stores the list. The second propose-completion rule provides completion suggestions for property access expressions, by fetching all property names and recording them in the dynamic rule. The control rule for content completion is called editor-complete. It first performs name analysis (extended with content completion support) and then fetches the value of the ContentProposals rule.

The editor descriptor specifies which control rule to use for content completion, and may specify a character se-

```
rename-top = alltd(propose-completion <+ rename)

propose-completion:
  Var(COMPLETION(p)) -> all-vars
  where all-vars := <all-keys-RenameId>;
        rules(ContentProposals: () -> all-vars)

propose-completion:
  PropAccess(e, COMPLETION(p)) -> p2*
  where Entity(x, p*) := <EntityDef>;
        p2*           := <map(property-name)> p*;
        rules(ContentProposals: () -> p2*)

editor-complete:
  (selected, position, ast, path, project-path) -> p*
  where editor-init;
        ast' := <declare-top> (<desugar-top> ast);
        p*   := <ContentProposals> ()
```

Figure 20. Content completion rules.

quence that automatically triggers content completion. For example, for NWL, content completion should be automatically triggered when the user types '.' or ':'.

4.8 Transformations, Code Generation, and Views

Based on the Stratego language, Spoofax supports transformations and code generation using concrete object syntax [50], allowing rules to be written using the concrete syntax of the language that is transformed as patterns. These patterns can be syntactically checked against the syntax of the language. The Stratego compiler converts the patterns to abstract syntax. Concrete syntax can be very effective for writing modular transformations that are divided into a series of small, separate transformation steps [23]. After transformation using concrete syntax, a pretty printer can produce text output from the result.

Concrete object syntax can be contrasted to template engines, where no further transformation steps are possible after code is rewritten (unless the resulting text is reparsed). Stratego also offers language support for template engine-like transformations, by providing indentation-safe string interpolation as a "quick and dirty" code generation solution. For the small NWL language we opted for this approach to code generation: after some desugaring and normalization, NWL is printed to Java, and no subsequent transformations are applied. Figure 21 shows code generation rules using string interpolation and the control rule for generating Java.

Transformations and code generation can be used as a basis for views. Views can be described using descriptors, and can have a title, a rule name, and a number of annotations:

```
builder: "Generate Java code" =
  generate-java (openeditor) (realtime)
builder: "Show abstract syntax" =
  generate-aterm (openeditor) (realtime) (meta)
```

This specification defines views to show Java code and the abstract syntax of a selection. The (openeditor) annotation specifies that an editor should be opened for the view (rather than just generating code and saving it a file). The (realtime) annotation specifies that the view should be updated in real-time as the source file is changed. Finally,

```
generate-java:
  (selected, position, ast, path, project-path) ->
  ($[[path].java], <to-java> selected)

to-java:
  Entity(x, p*) ->
  $[ class [x] {
       [p2*]
     }
   ]
  where p2* := <to-java> p*

to-java:
  Property(x, Type(t)) -> $[
    private [t] [x];
    public [t] get_[x] { return [x]; }
    public void set_[x] ([t] [x]) { this.[x] = [x]; }
  ]
```

Figure 21. Code generation rules.

the (meta) annotation specifies that the view should only be available to language engineers, not to end developers.

5. Experience

We have substantial experience in using Spoofax and languages created with Spoofax. Several languages are part of the standard distribution, and are used in the development of new languages:

- The SDF language. Developed before the Spoofax workbench, we described the static semantics of SDF using the idioms in this paper to support editor services such as views on syntax productions and content completion.
- The ESV editor descriptor language.
- The Stratego language. We described the static semantics to provide full-featured Stratego IDE support.
- The ATerm language [45], used for abstract syntax. In addition to the Spoofax specification, we added special support for views based on the source file of an ATerm file (e.g., allowing "generate Java code" to be applied to an ATerm created from an NWL file).

Spoofax has also been used, by ourselves and others, to develop a variety of new languages covering different domains. Some have been developed independently of our environment before Spoofax was available, and have used Spoofax to describe the syntax and static semantics for IDE integration. A selection:

- Acoda [48] is a tool set that uses DSLs to aid in data migrations as data models evolve. Given different versions of a data model, it can automatically derive textual, editable evolution traces that can be executed to perform data migration. The data model editor uses views to generate these traces and supports the full range of editor features for editing them.
- Aster [31], an attribute grammar language based on strategies for description of attribute propagation patterns.
- PIL [24], a Java-like, object-oriented programming language that targets multiple software platforms.

– Mobl, a DSL for web-based mobile phone applications. It generates HTML/Javascript code when files are saved, allowing them to be directly tested in a browser.
– NWL (shown in this paper) and WebDSL [51], a DSL for the Web. While WebDSL was developed using Stratego, it did not follow the idioms we presented in this paper that would allow it to fully integrate into and IDE. We are currently working on describing the full static semantics of the language and its extensions using this new style.
– An assembly language for Java bytecode, used mainly as a pedagogical tool for inspecting and creating Java class files, or to directly generate bytecode from other Spoofax languages. With over 200 instruction keywords it currently benefits greatly from syntactic content completion; semantic content completion could even improve the programming experience more based on the stack state at a given point in a program.

Observations To give a few observations about our experience, Spoofax allows for a much more agile development model than was previously possible with separate tools for syntax definition, meta-programming, and especially IDE development. As languages are developed they are directly usable in the IDE from the point that a syntax is defined. From there, developers can incrementally add additional functionality, from presentational editor services to semantic analyses and code generation. Not all languages listed above have yet reached the level of maturity where they incorporate the full set of services from Figure 2, but they still provide an improved user experience compared to a standard text editor. Using Spoofax also changed the deployment model of the languages: they can now be distributed as Eclipse plugins instead of separate command-line compilers, which tends to be much more appealing to “end developers.”

In our experience the Spoofax workbench significantly lowers the bar for creating a new language by providing an interactive environment for development and playing with it as it evolves. We have developed a number of experimental languages to try out new language concepts, and are now using it with great success in a course on model-driven software engineering. It allows students to experiment with language features and experience all aspects of language development in a unified environment. For a course on compiler construction, we will use the bytecode language to allow students to experiment with stack architectures and abstractions over a low-level language.

Customization Based on the Eclipse platform, Spoofax language plugins are open to various forms of extensions and integration with other plugins. In the WebDSL plugin we particularly made use of that, and integrated the plugin with the Eclipse Web Tools Platform for testing and deploying web applications. The plugin also provides its own wizard that configures database and deployment configuration. These customizations have been implemented as custom Java code, directly embedded in the plugin itself. The WebDSL plugin also integrates with the Acoda plugin: Acoda can extract data models from web applications, show these in a view, and then derive evolution traces by comparing data models.

Evolution As languages evolve with new insights, new constructs may be added and others removed, which can lead to incompatibilities with older programs. One way Spoofax assists in migrating older DSL programs to newer revisions of a language is through the notion of deprecated syntax. Syntactic constructs marked deprecated are displayed with a warning in the editor, and may be transformed to updated constructs using desugaring rules (Section 4.2) or by a migration transformation that processes these constructs and produces a migrated program. An area of future work is to provide tool support to assist in these migrations, much like Acoda can do for data models.

6. Implementation

The implementation of Spoofax is based on the Eclipse platform, allowing it to be used together with other, language-independent Eclipse plugins such as build management tools and other language-specific plugins.

For the definition of generic IDE components we make use of the IMP framework [7, 8], which greatly simplified the construction of these components by abstracting over the Eclipse API and guiding us through the development process using wizards. Languages developed in our environment maintain compatibility with IMP. In addition to the service descriptor DSLs, it is also possible to use Java-based implementations using the IMP framework to implement components of a language.

The language workbench integrates the language parametric editor service components based on IMP with the Spoofax libraries for parsing (JSGLR) and running Stratego (Stratego/J) [1]. As Spoofax/IMP — the official name of the workbench — integrates nearly all of the libraries of the Spoofax project, the workbench is also simply called the Spoofax language workbench.

6.1 Language-Parametric Editor Services

From an implementation point of view, only a small portion of an IDE or IDE plugin implementation is actually language-specific. A great part of the implementation of IDE components deals with accidental complexity that has little or no relevance to a particular language. IDE development frameworks such as IMP [7, 8] and TMF [56] simplify the implementation of these components by abstracting away from many of these details.

In Spoofax we implemented IDE components in a language-parametric fashion. Each component is written in Java, and interacts with the APIs for abstract syntax trees, tokens, text editor widgets, the file system, the parser and error recovery, etc. However, it does not include any specific knowledge of the language it is used for. Instead, the components interpret an editor descriptor that configures these language-specific parts. For each editor service, a factory class that reads the editor descriptor for that service, and instantiates the language-parametric implementation class for it. These classes use the standard Java collection classes such as hash tables to efficiently store and access the language configuration. For example, for the syntax highlighting service, we maintain hash tables that specify which formatting style to apply to which kind of character sequence. For parsing, we use a generated parse table that is dynamically loaded into the environment.

We use proxy classes to implement the Eclipse and IMP extension points for editor services, satisfying the static Eclipse/OSGi component model for loading plugins. These proxy classes invoke the corresponding factory class as services are first used, loading the parametrized implementation classes. When a language definition is changed, the proxy classes seamlessly load new services from the descriptors.

Parser technology and IDE components Key for rapid feedback in interactive environments is a parser that supports error recovery in order to parse files with syntax errors and incomplete programs as a programmer edits the text. As an example, consider the file at the left of our screenshot in Figure 4. This file is not syntactically correct, since the last production does not have a symbol it matches to. To still provide content completion, the parser still has to produce a sensible abstract representation for the editor services to operate on. In [11, 29] we described language-generic error recovery techniques that we use in the Spoofax language workbench to address this issue.

A number of editor services directly interact with the syntactic state of the parser. For instance, the bracket insertion service needs to determine if the cursor is not inside a string or comment terminal. If that is the case, and the user types an opening bracket `{`, the service should not automatically insert the closing bracket `}`. One way to determine whether the cursor is inside a string or comment is simply having a language-specific "blacklist" of syntactic constructs were bracket insertion should be disabled. However, to do this in a language-agnostic way, we had to take a different approach. Instead, we analyze the active production at the selected character, which can be determined from the parse tree. If it is a lexical production, we can examine the lexical character class that can be parsed by it. If the character class includes the bracket that would be inserted (e.g., `}`), then that indicates that the bracket would become part of the lexical. For these cases, closing bracket insertion is disabled.

Another editor service that interacts directly with the parser is content completion. As we discussed in Section 4.7, programs are often in a syntactically incorrect state when a completion proposal is demanded. For example, an expression `e.` may be a property access expression in the making. If content completion is triggered at that point, we insert a placeholder identifier at the point of the cursor, forming an expression `e.placeholder`, and parse the file again. This ensures that the expression is still parsed as a compound expression. Any missing semicolons, parentheses, etc. can be picked up by the regular error recovery rules. The placeholder also allows us to easily add an artificial `COMPLETION` constructor at the point where the placeholder appears, which is used for the content completion analysis of Section 4.7.

Interpretation versus compilation Using language-parametric editor services does not enforce an interpretative model. It is also possible to use language-parametric editor services with compiled Java classes that are dynamically loaded using classloaders. In fact, we use classloaders to load compiled Stratego specifications for semantic services. Using a compiled model can lead to improved runtime performance, but comes at the cost of compilation speed. For Stratego, developers can choose whether to interpret or compile specifications, depending on whether they want more agile development or if they want to distribute the plugin to "end developers." A similar approach could also be used for syntactic editor services, but we feel that the trade-off of compilation time versus performance for those services may be unacceptable: we have never seen performance problems for those services based on an interpretative model.

6.2 Semantic Services and Rewrite Rules

The Stratego language was originally compiled to C, but we developed a more flexible backend that compiles it to Java and interoperates with Eclipse. Where the C-based Stratego operates only on the ATerm data type of the ATerm library [45], the Java implementation is more flexible and can transform any tree structure that implements the `IStrategoTerm` interface. Using adapter classes, any Java data type can be made to implement it. This approach is called the program object model adapter (POM) approach. Previous experience with a Stratego interpreter on Java showed that this approach allows rule-based transformations to interoperate with Java-based frameworks such as a compiler frontends [28].

We use the POM adapter approach to transform trees that maintain position and layout information. By using a custom *factory class* used to construct new `IStrategoTerm` instances, we implemented a form of origin tracking [46] for Stratego. Origin tracking is key to concise, position-agnostic specification of transformations and analyses in Spoofax.

The term factory class is used to construct new terms by compiled and interpreted Stratego programs. Term factories have methods to *create* new terms given a constructor name/and or children or by parsing them from files [28]. We added methods to *replace* subterms of a term. Terms are functional, immutable data types, so by default these methods simply return a new term with the given subterms. In Spoofax, we return a new term that maintains a link to the original term and its children. We changed the implementation of the Stratego traversal operators `all`, `some`, and `one` to use the new "replace" methods. This way, all strategies that use these combinators (such as `alltd` and `collect-all`, shown in this paper) support origin tracking.

6.3 Editor Extensibility and Customization

Language workbenches provide an integrated development environment for building a language and IDE support, abstracting over low-level IDE implementation details. Still, it can be useful to have a "backdoor" to escape the environment and add features that (we) the developers of the workbench had not anticipated.

Spoofax language definitions can be extended using the standard Eclipse extension model. New components can be added to plugins simply by declaring them in the `plugin.xml` descriptor file and implementing them in Java. Since we base our implementation on IMP, helpful IMP wizards can even generate skeletal Java implementation for common editor services. Once declared, Java implementations of services can even override the Spoofax implementations. Drawbacks of adding services this way are that they do not provide the same abstractions as DSLs would, and that they tie developers to the standard Eclipse way of plugin development: custom, Java-based services can only be used in a secondary Eclipse instance.

7. Discussion and Related Work

Spoofax is a language workbench for textual domain-specific languages. Textual languages benefit from integration with standard, text-based version control systems and issue trackers, easy importing and exporting of files from other tools (avoiding vendor lock-in), files that are editable with other tools, and free text editing. Notable other tools for creating and using textual DSLs are the Meta-Environment [34, 44], EMFText [22], MontiCore [36], TCS [26], TEF [55], and Xtext [14]. A thorough comparison of a number of these tools has recently been provided by Goldschmidt et al [19] and by Pfeiffer and Pichler [40]. In our discussion we contrast these tools to Spoofax.

The Meta-Environment is a platform for language development, source code analysis, and source code transformation [34, 44]. It includes SDF, the ASF term rewriting language, and provides an IDE framework written in Java. The Meta-Environment derives basic syntax highlighting from SDF grammars. ASF tree-traversal may also be used to annotate the AST with coloring directives. ASF is also used to specify the typing rules of the language, and may include custom error messages, presented in a separate view. The IDE framework provides outlining but none of the other presentation, editing, or semantic services provided by Spoofax.

Compared to the Stratego language, ASF has limited expressive power. ASF is a pure term rewriting language, whereas Stratego adds strategies [52] and dynamic rules [4], which have been key for concise, reusable analyses in Spoofax. In Spoofax we use origin tracking for position-agnostic specification of editor services. Origin tracking was originally implemented as a prototype for ASF [46], but is no longer used in the current version.

EMFText [22], TEF [55], and TCS [26] take a different approach to syntax development than we do in Spoofax. Rather than combining the specification of concrete and abstract syntax into a single grammar, they start with the construction of a metamodel. TCS and EMFText have a generic concrete syntax that can be derived from a metamodel, at the cost of domain-specific notation. In EMFText, developers also have the option to write their own grammar with productions that map to elements of the metamodel. On the one hand, this approach allows the use of existing EMF metamodels for building new languages, which can be useful in certain cases. On the other hand, the approach is also less agile than using a syntax formalism that integrates abstract and concrete syntax: it introduces redundancies in the grammar, and requires the syntax to be maintained in two places. The three workbenches provide error markers, reference resolving, and content completion that can be customized using Java, but none of the presentation and editing services of Figure 2.

Xtext [14] is also based on EMF metamodels, but integrates the specification of concrete syntax and metamodel into a single grammar specification. Xtext was originally part of openArchitectureWare [13], and used the OCL-like Xtend language for model transformations and for the specification of problem markers using constraints and for reference resolving. Recently, Xtext has been reimplemented using TMF [56]. It now relies on grammar annotations and uses Java code to describe the static semantics of languages and all but the most basic editor services.

The abstract representation of programs in Spoofax is based on trees, where dynamic rules superimpose graph structures. In contrast, EMF is based on graphs. EMFText and Xtext allow annotations in grammar productions to specify simple use-def relations between productions that introduce back-edges to the abstract syntax tree, effectively producing graphs. This approach of syntactic name resolution only works for names with a global scope (or lexical scopes in the case of EMFText). It is inadequate for compound expressions (e.g., `blog.author`). EMFText and Xtext resort to Java code to specify these relations. In Spoofax, the name analysis is fully specified in a rule-based Stratego specification. An interesting future step for Spoofax could be to map abstract syntax trees and dynamic rules to EMF models, and express constraints over such models with Stratego.

MontiCore [36] generates custom Java classes to represent the abstract syntax. It supports basic presentational services, which are specified as grammar properties. Syntax coloring is specified as lists of keywords to highlight. Pre-defined (Java-style) comments are supported. Folding is specified by a list of non-terminals. For semantic editor services, MontiCore grammars specify events, which may be specialized with user-defined Java classes.

Interestingly, with the exception of the Meta-Environment and TEF, all tools that we described generate ANTLR parsers. TEF uses a the RunCC parser generator. ANTLR's LL(k) or LL(*) parsers cannot cope with left recursion in grammars. Likewise, RunCC's LR(k) parsers are limited to a subset of the context-free grammars. This means that they are not closed under composition, which means that adding extensions to a grammar or reusing grammars can introduce conflicts in the parser [32]. These parsers also rely on using a single, separate scanner, which means that adding extensions with a different lexical syntax is not possible. In contrast, Spoofax and the Meta-Environment use SDF to specify grammars and generate scannerless generalized LR (SGLR) parsers. Based on SGLR parsing, SDF grammars can be freely composed, allowing for embedded languages and language extensions.

One area in which Spoofax excels compared to other tools is in supporting agile language design and development. Languages can be incrementally developed and changes can be dynamically loaded into the environment. Spoofax automatically derives syntactic editor services from the syntax definition. Editor services can then be selectively customized as desired. In contrast, the other tools often provide generic syntax highlighting for keywords and hardcoded default symbols such as strings, but do not derive highlighting or other services based on analysis of the grammar. Of the other tools, only the Meta-Environment and TCS have the ability to dynamically reload language definitions without requiring the editor environment to be relaunched. In the case of the Meta-Environment, this comes at the cost of the idiosyncrasies of a custom IDE environment rather than integration into a common language platform such as Eclipse. TCS integrates into Eclipse, but is primarily designed for adding a textual syntax to languages that have a model-based principal representation. While it allows free text editing, it does not allow the same flexibility in language design as other workbenches where language engineers can write their own grammar.

We use the Stratego language for analyses, transformations, and code generation. Other workbenches use Java classes for analysis, sometimes combined with grammar annotations, Java visitors for transformations, and template engines for code generation. Based on strategies and rewrite rules, Stratego concisely specify analyses and transformations. Providing a choice of concrete object syntax (ensuring syntactic correctness) [50] and string interpolation, Stratego also provides a flexible solution for code generation.

Other notable meta-programming languages include ASF+SDF [12], JastAdd [20], Rascal [35], and TXL [10]. They provide limited IDE support at this point, although the developers of JastAdd and Rascal are actively working on IDE support dedicated to their meta-programming languages. However, much like the original Spoofax Stratego editor [27], they do not provide a language workbench solution for languages developed with them. Like Stratego, ASF+SDF and TXL support a form of concrete object syntax, but they do not support string interpolation. Only Rascal supports both forms of code generation. JastAdd relies on Java visitors on the abstract syntax tree to generate code.

8. Open Issues and Future Work

In this paper we showed that the Stratego language can be used for concise specifications of analysis, transformations, and code generation. Still, many other meta-programming languages exist that each have their own merits and uses. We believe that Spoofax has the potential to become a common, open platform for hosting multiple meta-programming languages. Spoofax defines a lightweight, technology-agnostic interface between editor services and semantic analyses. Analysis specifications in other meta-programming languages could follow the same interface, and to some degree may follow the same idioms for reuse, allowing them to implement the same set of editor services. One constraint is the representation of the abstract syntax, which may require marshalling to another form. Another is our use of origin tracking in the specification of analyses. Without origin tracking, semantic specifications would have to explicitly ensure that analysis results contain position information.

In previous work, we applied strategic programming in the field of attribute grammars, enabling high-level, declarative specifications of semantic analyses in the Aster language [31]. Based on Aster's reusable attribute propagation patterns for name, type, and flow analysis, we would like to investigate how Aster specifications can be integrated into Spoofax and used to specify IDE services.

Another area of future work is in providing tool support for first-class language components. Based on the modular SDF syntax formalism and SGLR parsing, it is possible to decompose languages into separate, reusable components [5]. As an example, WebDSL [51] reuses the syntax definition of HQL for queries. Such compositions still require much manual work, as Spoofax provides only limited tool support for these forms of reuse. One ideal scenario may be that users of a language could just pick and match the language features they need, and that the environment composes them. Composition at the semantic level poses additional challenges. Based on a uniform type system and a single host language, it is possible to combine language components. (Notably, MPS [25] relies on these properties.) However, if a different host language or type system is used, reuse is currently limited to the syntactic level.

Spoofax provides support for a significant number of editor services found in modern IDEs, and provides a solid foundation for implementing new services, based on background parsing and reuse of semantic analyses. Users can currently add additional components using Java, but we would like to support more services using DSLs in the future. Services that we would particularly like to support are refactoring and debugging. In current IDE implementations, these services are especially demanding to implement, requiring sophisticated analyses for refactoring, and interaction with the runtime state of applications for debugging.

Refactorings are highly language-specific transformations, but they often take common forms: rename "something," extract "something," etc. We expect that it would be possible to describe preconditions, postconditions, and invariants of these forms of refactorings in a uniform way based on our idioms for name and type analysis. Based on the Stratego language, and using origin tracking for layout-agnostic transformations, we hope to find ways to efficiently and succinctly express refactorings.

For debugging, the runtime state of domain-specific languages can take many different forms. The IMP platform only provides rudimentary support for of languages that generate Java code [8]. While we could also add support for debugging of DSLs that generate Java, we are also interested in making it easier to describe debuggers in a more technology-agnostic way, and in finding general patterns to efficiently construct debuggers for arbitrary DSLs.

9. Conclusion

Modern IDEs increase developer productivity by incorporating many different kinds of editor services specific to the syntax and semantics of a language. They assist developers in understanding and navigating through the code, they direct developers to inconsistent or incomplete areas of code, and they even help with editing code by providing automatic indentation, bracket insertion, and content completion. As a consequence, developers that have grown accustomed to these services are growing less accepting of languages that do not have solid IDE support.

To efficiently develop languages with IDE support, Spoofax supports selective, incremental development of editor services that can be dynamically loaded, evaluated, and tuned in the same environment. Using high-level languages to specify the syntax and semantics of a language, it provides a language development solution that greatly increases productivity of language engineers in building language and IDE components compared to using handwritten components or separate language engineering tools.

Acknowledgements This research was supported by NWO/JACQUARD projects 612.063.512, *TFA: Transformations for Abstractions*, and 638.001.610, *MoDSE: Model-Driven Software Evolution*. We would like to thank Karl Trygve Kalleberg for his many contributions to the Spoofax project; the development teams of IMP, SDF, (J)SGLR, and Stratego/XT for their valuable efforts; Maartje de Jonge and Emma Söderberg for their contributions to error recovery with JSGLR; Rob Vermaas, Bernhard Merkle, and the anonymous reviewers for suggestions for this paper; and finally our users for providing continuous feedback and coming up with interesting new ideas.

References

[1] The Spoofax project. `http://www.spoofax.org/`.

[2] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. *SIGPLAN Not.*, 24(2):14–24, 1989.

[3] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Sci. of Comp. Programming*, 72(1-2):52–70, June 2008. Special issue on experimental software and toolkits.

[4] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69(1–2):123–178, 2006.

[5] M. Bravenboer and E. Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *OOPSLA*, pages 365–383, 2004.

[6] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework*. Addison-Wesley, 2004.

[7] P. Charles, R. M. Fuhrer, and S. M. Sutton, Jr. IMP: a meta-tooling platform for creating language-specific IDEs in Eclipse. In *ASE 2007*, pages 485–488, 2007.

[8] P. Charles, R. M. Fuhrer, S. M. Sutton, Jr., E. Duesterwald, and J. Vinju. Accelerating the creation of customized, language-specific IDEs in Eclipse. In *OOPSLA 2009*. ACM, 2009.

[9] S. Cook, G. Jones, S. Kent, and A. C. Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison Wesley, 2007.

[10] J. R. Cordy, C. D. Halpern-Hamu, and E. Promislow. TXL: a rapid prototyping system for programming language dialects. In *Conf. on Comp. Languages*, pages 280–285. IEEE, 1988.

[11] M. de Jonge, E. Nilsson-Nyman, L. C. L. Kats, and E. Visser. Natural and flexible error recovery for generated parsers. In *SLE*, 2010.

[12] A. v. Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Sci. Publ. Co., 1996.

[13] S. Efftinge et al. openArchitectureWare User Guide. Version 4.3. Available from `http://www.eclipse.org/gmt/oaw/doc/4.3/html/contents/`, April 2008.

[14] S. Efftinge and M. Voelter. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, 2006.

[15] M. Fowler. A language workbench in action - MPS. `http://martinfowler.com/articles/mpsAgree.html`, 2005.

[16] M. Fowler. Language workbenches: The killer-app for domain specific languages? `http://martinfowler.com/articles/languageWorkbench.html`, 2005.

[17] M. Fowler. PostIntelliJ. `http://martinfowler.com/bliki/PostIntelliJ.html`, 2005.

[18] M. Fowler. A pedagogical framework for domain-specific languages. *IEEE Software*, 26:13–14, 2009.

[19] T. Goldschmidt, S. Becker, and A. Uhl. Classification of concrete textual syntax mapping approaches. In *ECMDA-FA 2008*, volume 5095 of *LNCS*, pages 169–184. Springer, 2008.

[20] G. Hedin and E. Magnusson. JastAdd: an aspect-oriented compiler construction system. *Sci. Comput. Program.*, 47(1):37–58, 2003.

[21] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF: Reference manual. *SIGPLAN Not.*, 24(11):43–75, 1989.

[22] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Derivation and refinement of textual syntax for models. In *ECMDA-FA*, pages 114–129, 2009.

[23] Z. Hemel, L. C. L. Kats, D. M. Groenewegen, and E. Visser. Code generation by model transformation. A case study in transformation modularity. *Softw. and Syst. Modeling*, 2009.

[24] Z. Hemel and E. Visser. PIL: A platform independent language for retargetable DSLs. In *SLE*, 2010.

[25] JetBrains. Meta programming system. `https://www.jetbrains.com/mps`.

[26] F. Jouault, J. Bézivin, and I. Kurtev. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In *Generative and Component Engineering (GPCE'06)*, pages 249–254. ACM, 2006.

[27] K. T. Kalleberg and E. Visser. Spoofax: An interactive development environment for program transformation with Stratego/XT. In *Workshop on Language Descriptions, Tools, and Applications (LDTA 2007)*, pages 47–50, 2007.

[28] K. T. Kalleberg and E. Visser. Fusing a transformation language with an open compiler. In *Workshop on Language Descriptions, Tools, and Applications (LDTA 2007)*, volume 203 of *ENTCS*, pages 21–36. Elsevier, April 2008.

[29] L. C. L. Kats, M. de Jonge, E. Nilsson-Nyman, and E. Visser. Providing rapid feedback in generated modular language environments. Adding error recovery to scannerless generalized-LR parsing. In *OOPSLA*, pages 445–464, 2009.

[30] L. C. L. Kats, K. T. Kalleberg, and E. Visser. Domain-specific languages for composable editor plugins. In *Workshop on Language Descriptions, Tools, and Applications (LDTA 2009)*. Elsevier, April 2009.

[31] L. C. L. Kats, A. M. Sloane, and E. Visser. Decorated attribute grammars. Attribute evaluation meets strategic programming. In *Conference on Compiler Construction (CC 2009)*, volume 5501 of *LNCS*, pages 142–157. Springer, March 2009.

[32] L. C. L. Kats, E. Visser, and G. Wachsmuth. Pure and declarative syntax definition: Paradise lost and regained. In *Onward!*, 2010.

[33] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling. Enabling Full Code Generation*. John Wiley & Sons, Inc., 2008.

[34] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, 1993.

[35] P. Klint, T. van der Storm, and J. Vinju. Rascal: a domain specific language for source code analysis and manipulation. In *SCAM*, pages 168–177, 2009.

[36] H. Krahn, B. Rumpe, and S. Völkel. Monticore: Modular development of textual domain specific languages. In *TOOLS*, pages 297–315, 2008.

[37] M. F. Kuiper and J. Saraiva. Lrc - a generator for incremental language-oriented tools. In *Compiler Construction (CC'98)*, pages 298–301, London, UK, 1998. Springer-Verlag.

[38] M. Mernik, J. Heering, and A. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, 37(4):344, 2005.

[39] N. Nystrom, M. Clarkson, and A. Myers. Polyglot: An Extensible Compiler Framework for Java. *Compiler Construction (CC'03)*, 2622:138–152, Apr. 2003.

[40] M. Pfeiffer and J. Pichler. A comparison of tool support for textual domain-specific languages. In *Workshop on Domain-Specific Modeling*, pages 1–7, 2008.

[41] T. Reps and T. Teitelbaum. The synthesizer generator. *SIGSOFT Softw. Eng. Notes*, 9(3):42–48, 1984.

[42] S. Saunders, D. K. Fields, and E. Belayev. *IntelliJ IDEA in Action*. Manning, 2006.

[43] C. Simonyi. The death of computer languages, the birth of Intentional Programming. Tech. report, MS Research, 1995.

[44] M. Van den Brand, A. Van Deursen, J. Heering, H. De Jong, et al. The Asf+Sdf Meta-Environment A Component-Based Language Development Environment. In *Compiler Construction*, volume 44 of *LNCS*, pages 365–370. Springer, 2001.

[45] M. G. J. van den Brand, H. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

[46] A. van Deursen, P. Klint, and F. Tip. Origin tracking. *Journal of Symbolic Computation*, 15(5/6):523–545, 1993.

[47] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.

[48] S. Vermolen and E. Visser. Heterogeneous coupled evolution of software languages. In *MoDELS*, pages 630–644, 2008.

[49] E. Visser. A family of syntax definition formalisms. Technical Report P9706, Programming Research Group, University of Amsterdam, July 1997.

[50] E. Visser. Meta-programming with concrete object syntax. In *GPCE*, pages 299–315, 2002.

[51] E. Visser. WebDSL: A case study in domain-specific language engineering. In *GTTSE*, pages 291–373, 2007.

[52] E. Visser, Z.-E.-A. Benaissa, and A. P. Tolmach. Building program optimizers with rewriting strategies. In *ICFP*, pages 13–26, 1998.

[53] W. Waite and G. Goss. Compiler construction. 1984.

[54] M. P. Ward. Language-oriented programming. *Software — Concepts and Tools*, 15(4):147–161, 1994.

[55] Textual Editing Framework (TEF). `http://www.informatik.hu-berlin.de/sam/meta-tools/tef`.

[56] Textual modeling framework (TMF). `http://www.eclipse.org/modeling/tmf/`.

[57] The WAtson Libraries for Analysis. `http://wala.sourceforge.net/`.