6 An Extensible Transformation Language

This chapter introduces new compiler and runtime infrastructure that turns Stratego into an extensible transformation language. The extensible variant of Stratego is used throughout this dissertation as a platform for experimenting with language extensions and abstractions for language-independent transformations. The extensible compiler reuses most of the Stratego compiler and the Stratego library. It is a strict – or conservative – extension of the Stratego infrastructure. The compiler is complemented by a versatile transformation runtime designed to abstract over different term representations. The runtime supports dynamic loading of transformation components during execution. These components are expressed using a new and light-weight component architecture for self-contained transformation components.

6.1 An Extensible Compiler

The primary design goal of the extensible compiler, called the MetaStratego compiler, is to support the development of language extensions for Stratego. Figure 6.1 illustrates examples of language extensions under development with this compiler. Developers of transformations may add their own extensions (illustrated with the stippled box containing `your-stratego`) as part of a specific transformation project.

6.1.1 Declaring Syntax and Assimilator

The extensible compiler enables programmers to plug in language extensions at compile time. A programmer may do this in either of two ways. Each Stratego module is implemented in a `.str` file, for example `foo.str`. By creating a so-called meta file, it is possible to give instructions to the compiler about how to treat the contents of `foo.str`. The following is an example meta file, `foo.meta`:

```
Meta([
  Syntax("AspectStratego"),
  Assimilator("assimilate-aspects")
```

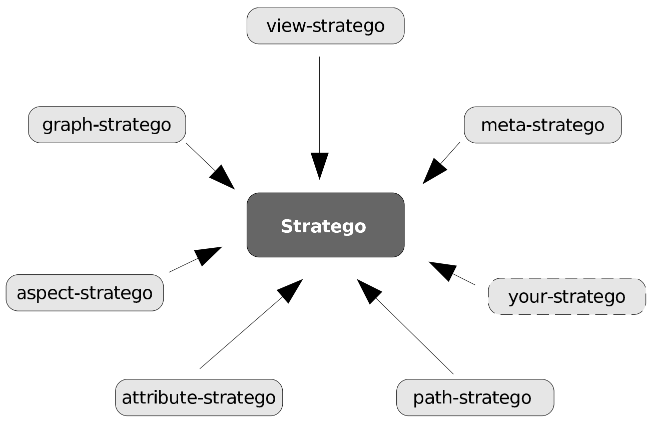

Figure 6.1: Language extensions for Stratego implemented using MetaStratego. This dissertation discusses GraphStratego and AspectStratego. The reason for the depicted MetaStratego extension is explained in Section 6.1.1.

```
  ])
```

This file must always accompany `foo.str`. It declares to the MetaStratego compiler that the `AspectStratego` syntax must be used to parse the file `foo.str`. The meta file also declares that the AspectStratego language extension is to be translated (assimilated) into plain Stratego using a transformation component called `assimilate-aspects`.

When the file `foo.str` is compiled, the MetaStratego compiler will search the include path for the `AspectStratego` syntax and the `assimilate-aspects` assimilator program. For convenience, it is also possible to specify the language extension inside the module. This is done in a separate meta section:

```
module foo
meta
  syntax = "AspectStratego"
  assimilator = "assimilate-aspects"
imports
  ...
```

The meta section must always appear immediately after the module declaration. A small pre-processing step will read the top of the file and extract the meta information. The choice between meta files and meta sections is a matter of programmer preference.

Extending Stratego with a given language extension results in an *extended* Stratego language. For example, extending Stratego with support for aspects results in AspectStratego. The term *plain Stratego* will be used to refer to Stratego without any language extensions.

A short note on the meta section may be warranted. Meta sections are not part of plain Stratego. Support for meta sections is due to a tiny syntax extension provided by the MetaStratego compiler. It will by default parse all source files using this MetaStratego syntax. Technically speaking, all language extensions are therefore extensions of MetaStratego, not plain Stratego.

6.1.2 Language Extensions

The language extensions presented in this dissertation are all composed of two parts; a *notational component*, which extends the Stratego syntax, and a *transformation component* which provides the semantics of the extension. A language extension may be deployed separately from the compiler. The build system of a given project must declare the relevant paths of all necessary syntax extensions to the MetaStratego compiler.

The notational component of a language extension reuses the syntax extension mechanisms provided in [Vis02, BV04]. By composing grammar modules for the syntax extensions with the base grammar for Stratego[1], an extended Stratego language is obtained. Programs in the extended language are parsed with the parse table generated from this extended grammar. The compiler front-end will produce a corresponding extended AST which contains extension-specific nodes.

The semantics of the extension is expressed as a transformation from the extended AST into the plain Stratego AST. These transformation are called assimilators [BV04] because they assimilate (embed) the extension into plain Stratego. Developers implementing assimilators make use of the standard Stratego library, the Stratego compiler library and the MetaStratego compiler library as shown in Figure 6.2.

6.1.3 Compiler Pipeline

The MetaStratego compiler pipeline is depicted in Figure 6.3. The MetaStratego compiler reuses most of the standard Stratego compiler, but supplements it with a some new functionality and extension points. The standard compiler provides a mechanism for embedding concrete syntax patterns into the transformation program [Vis02]. It essentially provides the compiler user with an option to specify which grammar should be used to parse a given source file. MetaStratego relies on this mechanism for providing the additional syntax offered by the language extensions.

Many assimilators may be formulated so that they only interact with the compiler at one point: in the `process-metas` stage. All the language extensions proposed

[1]Strictly speaking, the grammar for MetaStratego which, except for meta section support, is identical to the Stratego grammar.

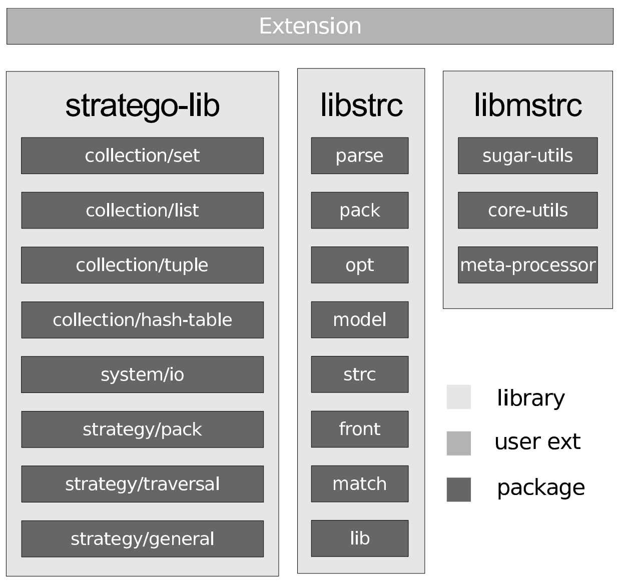

Figure 6.2: Transformation libraries available for implementing language extensions. `stratego-lib` is the Stratego standard library, `libstrc` is the Stratego compiler library and `libmstrc` is the MetaStratego compiler library.

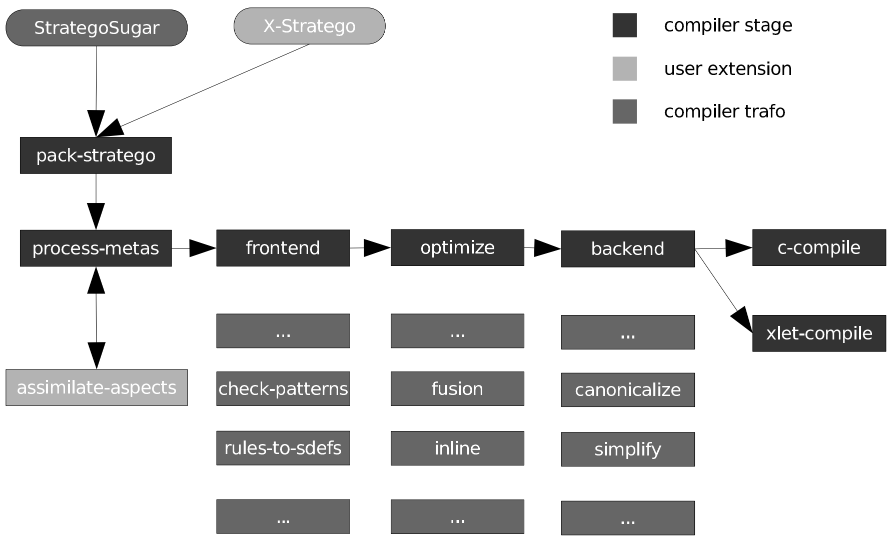

Figure 6.3: The extensible MetaStratego pipeline. The component `assimilate-aspects` is the assimilator for the AspectStratego language described previously. Using the extension points, the assimilators can hook into the various stages of the compiler.

```
signature StrategoCore
sorts Env Strat Var Pat
ops
  GChoice : Env × Strat × Strat × Strat → Env
  Match : Env × Pat → Env
  Build : Env × Pat → Env
  Scope : Env × List(Var) × Strat → Env
  Seq : Env × Strat × Strat → Env
  One : Env × Strat → Env
  All : Env × Strat → Env
  Call : Env × Var → Env
  Fail : Env → Env
  Id : Env → Env
```

Figure 6.4: Signature for the StrategoCore language. The semantics of most operators were described in Chapter 3.

in this dissertation are able to translate the entire extension into plain Stratego before the front-end stage. This may not always be possible. For example, additional optimisation opportunities may arise as a result of the extension. Plugging into the `optimize` stage may therefore be desired. For this reason, MetaStratego pipeline exposes new extension points into the compiler pipeline. New transformations may be added before or after each compiler stage shown in Figure 6.3.

A given extension may attach several assimilators to the various stages in the compiler. When a given extension point is reached, all registered assimilators for that stage will be executed. A limitation of the current extension scheme is that all extensions must be serialisable. That is, the order of all assimilators must be linearised and they must be executed sequentially. Multiple and co-existing language extensions are still possible, but they are difficult to use because the user must ensure that the assimilators are listed in the correct order in the meta file (or section).

6.1.4 StrategoCore

The output at the end of the backend stage in Figure 6.3 is a program in a minimal core language called StrategoCore. This language, specified in Figure 6.4, is the very close to the barest minimum required for implementing System S. Translating plain Stratego into StrategoCore is a stepwise process. It is complete at the end of the front-end. Various optimisations in the `optimize` stage, such as optimisation of pattern matching, are performed on the core format.

Both the MetaStratego and the Stratego compiler have an option to output programs in the core format. The core format is independent of any operating environment or hardware architecture. It is therefore a good candidate format for representing Stratego programs in a portable manner which can be loaded dynamically on a sufficiently capable transformation runtime.

6.2 An Extensible Runtime

The extensible compiler is complemented with an extensible runtime for Stratego. This runtime is designed around the general term library interface described in Chapter 4. It allows the runtime to perform rewriting on any data structure which can be mapped to this interface. The primary motivation for constructing this runtime was to enable large-scale reuse of transformation systems by plugging them into development environments, compilers and other language infrastructures.

To support these experiments, the runtime has been implemented in Java. Java is the *lingua franca* of modern software development and a substantial collection of front-ends and compilers for various languages have been implemented in Java, for C, C++, SQL, Python, Ruby, Fortress and others. Integrated development environments, like NetBeans and Eclipse, are frequently implemented in Java.

An additional feature of the runtime is its ability to load Stratego programs (in the StrategoCore format) dynamically at runtime. This feature makes it possible to extend a transformation system dynamically with new functionality. This is used in the Stratego development environment described in Chapter 9.

6.2.1 Design

The runtime provides an interpreter for StrategoCore files and extension facilities for plugging in new program object models and foreign function libraries. The interpreter is called Stratego/J, but is sometimes referred to as the MetaStratego runtime. An example instantiation of this runtime is shown in Figure 6.5.

In this diagram, two program object model adapters have been plugged into the interpreter (`org.spoofax.interpreter`). Program object models for the Eclipse Compiler for Java are adapted by the `org.spoofax.interpreter.adapter.ecj`. The component `org.spoofax.interpreter.adapter.aterm` adapts the ATerm library. Additional foreign functions for calling into the Eclipse platform – e.g. for opening windows in Eclipse and hooking into menus – are provided by `org.spoofax.library.eclipse`.

The runtime is supplemented with a scannerless GLR parser implemented in Java by the author. This parser, called jsglr, is compatible with the SGLR [Vis97] parser, developed at CWI, The Netherlands. It enables complete systems with the complete transformation cycle (parse-transform-unparse) to run on the Java platform and be plugged into development environments.

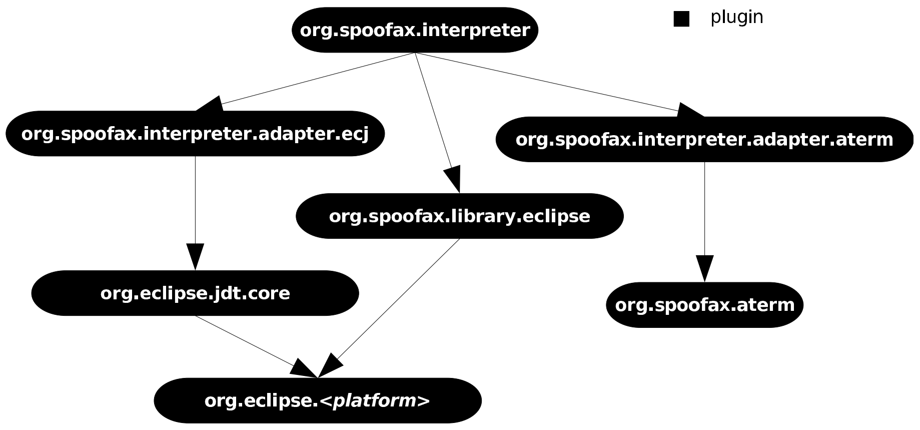
■ plugin
org.spoofax.interpreter
org.spoofax.interpreter.adapter.ecj
org.spoofax.interpreter.adapter.aterm
org.spoofax.library.eclipse
org.eclipse.jdt.core
org.spoofax.aterm
org.eclipse.<platform>

Figure 6.5: Components of the Stratego/J runtime.

6.2.2 Implementation

The implementation scheme of the interpreter engine is largely unremarkable. It follows the semantics of System S [BvDOV06] closely. A choice point stack [Mor98] is used to support backtracking. Some minor optimisations have been added to keep the stack depth at a minimum by throwing away stack frames whenever possible. This is an important consideration because Stratego programs are often deeply recursive.

6.2.3 Performance

The current performance of the interpreter is significantly slower, around 10–20 times, than that of natively compiled Stratego code. Additional optimisations are possible, especially in the pattern matching code, but it is unlikely that the interpreted code will ever perform on par with the natively compiled output of the Stratego compiler. The interpreter is capable of executing the Stratego (and the MetaStratego) compiler, thereby completely hosting Stratego on the Java Virtual Machine (JVM).

An attractive future direction for the Stratego compiler would be to add support for compiling strategy definitions to Java bytecode. It may also be worthwhile to experiment with an incremental (just in time) compilation scheme so that compilation only happens for frequently used strategies.

6.3 Light-Weight and Portable Transformation Components

The Stratego/XT environment provides a conceptually attractive component system called XTC [BKVV06], but this system suffers from some rather severe performance issues. It is based around the concept that every component runs in its own process. These processes are composed using (UNIX) pipes. Each component is a console application that reads its input from the standard input stream and returns its result on the standard output stream. All data is thus serialised from one component to another via a the operating system file API and this incurs considerable performance penalties both because of serialisation and because of process startup. A significant advantage of the model is that composition can be done at deployment time, instead of at compile-time – components may easily be swapped in and out after they are compiled.

Because of its performance limitations, XTC is currently being phased out of Stratego/XT and replaced by more traditional, dynamically linked libraries. This resolves the performance issues, but all components must be linked into the transformation pipeline statically. Also, the build processed is complicated slightly because of this fact since all components must be accounted for ahead-of-time. There is still a place for easily deployable transformation components, however, and especially when integrating transformation systems with interactive environments. This is why the light-weight transformlet component system presented next was designed. It should not be considered a full replacement for XTC, however.

6.3.1 Transformlets

A *transformlet* is a small, self-contained transformation component produced from a Stratego module (program) using the MetaStratego compiler. Figure 6.6 shows an example transformlet. A transformlet must declare information about itself, called meta information. It may also declare that it is capable of extending specific hooks in the environment that it will be loaded into.

The meta information must be declared using the strategy `xlet-meta-info`, as shown in the example. Deployment of the transformlet requires this meta information for packaging the transformlet into a deployable transformation component called an `.xlet` file.

When a transformlet (an `.xlet` file) is loaded into the runtime environment it undergoes *activation*. The runtime actives a transformlets by querying the transformlet for the presence of hooks. This is done by calling a strategy provided by the transformlet named `xlet-define-hooks`. The developer of a transformlet must know the names of the available hooks in the deployment environment when the transformlets is written. Extension of hooks are declared in the strategy `xlet-define-hooks`. This strategy will be invoked when the transformlet is loaded into its environment and must produce a list of tuples. Each tuple contains the name of a hook and the strategy that should be invoked when the hook is invoked.

Using this hook mechanism, it is possible to express open-ended callbacks. This is useful for interactive environments. When the runtime is asked to execute actions for a given hook, all registered strategies will be executed. This mechanism is used internally by the interactive development environment described in Chapter 9 to provide extensibility via user-written scripts.

6.3.2 Implementation

Each transformlets is composed of two parts: the *package descriptor* and the *core program*. It is represented as a (possibly compressed) ATerm. Consider the code in Figure 6.6. A valid transformlet is obtained by first compiling this module into a StrategoCore file. This file is a term representing the entire transformation program. A program called `xlet-make` is then applied to this term. `xlet-make` extracts and removes the meta information embedded in the core file. That is, the strategy `xlet-meta-info` is removed from the StrategoCore file. Finally, the extracted meta information and modified StrategoCore file are composed into a term corresponding to the signature in Figure 6.7. The root of the transformlet is an `XLet` term. The `Specification` subterm is the top-level StrategoCore term; this is where the modified program is placed. The meta information becomes the package descriptor.

6.4 Summary

This chapter described an extensible compiler for the Stratego transformation language together with a versatile and extensible transformation runtime. The platform supports dynamic loading of light-weight and portable transformation components called transformlets.

The basic infrastructure introduced in this chapter has been used as an experimentation platform for all the abstractions and case-study prototypes presented in this dissertation.

```
module example
imports
  spoofax/transformlet/-

strategies

  xlet-meta-info =
  !XLet(
     PackageDescriptor(
         Name("example")
       , Version("0.1.0")
       , [APIVersion("0.1.0")]
       , Dependencies([PackageRef(Name("spoofax"), APIVersion("0.1.0"))])
       , [ License("GPL-2")
         , Author(
             AuthorName("Karl Trygve Kalleberg")
           , AuthorEmail("karltk@stratego.org")
           )
         ]
       )
    , None
    )

  xlet-define-hooks = ![("hello-action", "hello")]

  hello = <debug> "Hello, World"
```

Figure 6.6: Example transformlet. The meta information is defined by `xlet-meta-info`. The strategy `xlet-define-hooks` defines that if the hook `hello-action` is invoked, the `hello` strategy should be called.

```
signature
  sorts
   XLet Name Version APIVersion Dependencies VersionRange
   MetaInfo AuthorName PackageDescriptor
  constructors
    XLet : PackageDescriptor × Specification → XLet
    PackageDescriptor: Name × Version × APIVersion × Dependencies ×
List(MetaInfo) → PackageDescriptor
    Name : String → Name
    Version : String → Version
    APIVersion : String → APIVersion
    Dependencies : List(PackageRef) → Dependencies
    PackageRef : Name × List(APIVersoin) → PackageRef
    VersionRange : Version × Version → VersionRange

    Author : AuthorName × AuthorEmail → MetaInfo
    AuthorName : String → AuthorName
    AuthorEmail : String → AuthorName

    License : String → MetaInfo
    Homepage : String → MetaInfo
    BugTracker : String → MetaInfo
```

Figure 6.7: Signature for the transformlet programs.