8

Language Extensions as Transformation Libraries

The language abstractions proposed on this dissertation are provided in form of extensions to Stratego that were built as transformation libraries using the MetaStratego framework. There is little specific to Stratego that makes it an inherently extensible language. It is, however, extremely well suited for implementing language extensions.

This chapter contains a case study which serves to illustrate that some of the experience gained while conducting the primary investigation – language-independent software transformations – is also easily applicable to language extension in general. The language extensions proposed in this dissertation are formulated as transformation libraries complemented with a convenient notation, in the form of a syntax extension to Stratego, using some of the techniques introduced in [BV04]. The same techniques can be applied easily to any language. To further illustrate this, the author implemented a small language extension to the small toy language called TIL [Vis05b] for handling alerts.

The underlying extension technique was subsequently presented and discussed in the paper *DSAL = library+notation: Program Transformation for Domain-Specific Aspect Languages* [BK06] written with Anya Bagge. The paper distills and discusses the principal details of this approach to language extension. It follows the tradition of language embedding suggested in [Vis02, BV04] but focuses the non-local effects of the language embeddings due to the cross-cutting nature of the embedded language.

This chapter is a verbatim reprint of the above-noted paper with the exception of some minimal formatting changes.

8.1 Abstract

Domain-specific languages (DSLs) can greatly ease program development compared to general-purpose languages, but the cost of implementing a domain-specific language can be prohibitively high compared to the perceived benefit. This is more

pronounced for narrower domains, and perhaps most acute for domain-specific aspect languages (DSALs).

A common technique for implementing a DSL is writing a software library in an existing programming language. Although this does not have the same syntactic appeal and possibilities as a full implementation, it is a technique familiar to most programmers, and it can be done cheaply compared to developing a full DSL compiler. Subsequently, the desired notation may be implemented as a simple syntactic preprocessor. The cross-cutting nature of DSALs, however, makes it difficult to encapsulate these in libraries.

In this paper, we show a technique for implementing a DSAL as a *library+notation*. We realize this by implementing the library in a program transformation system and the notation as a syntactic extension of the subject language. We discuss our experience with applying this technique to multiple kinds of DSALs.

8.2 Introduction

The implementation of domain-specific abstractions is usually done by way of libraries and frameworks. Although this provides the semantics of the domain, it misses out on good notation and many optimisation opportunities. Implementing domain specific languages by adding notation (syntax) to a library, and then programming a simple compiler that translates from the notation into equivalent library calls is an easy and powerful technique, which is cost-effective in many larger domains. Both the libraries and the simple compiler can be implemented in general purpose languages without too much effort, and it is important to note that the library need not be implemented in the same language as the compiler. If the "library" language supports syntax macros, like Scheme [DHB92], or has a sufficiently powerful meta-programming facility, like C++ templates [AG05], the translation task may be accomplished through the inherent meta-programming constructs of this language. Otherwise, a stand-alone preprocessor is commonly used. For example, adding complex numbers or interval arithmetic to Java, with an appropriate mathematical notation, can be accomplished by writing or reusing a Java library, and writing a simple translator from the mathematical notation into OO-style calls. The approach of adding notation to (object-oriented) libraries was explored in the MetaBorg project [BV04], where the subject language Java was extended in various ways using Stratego as the meta-programming language.

For domain-specific aspect languages, the translation story is different. Behind the notation visible to the programmer lie cross-cutting concerns which may reach across the entire program, possibly requiring extensive static analysis to resolve. The straight-forward translation scheme into library calls for the subject language is not applicable as we are no longer dealing with basic macro expansion. Instead, we shall

view aspects as meta-programs that transform the code in the base program. These meta-programs may be implemented with transformation libraries in a transformation language (which may be different from the subject language). This allows us to consider DSALs as syntactic abstractions over transformation libraries, analogous to the way DSLs are syntactic abstractions over base libraries in the subject language. That is, we do not translate the DSAL notation into library calls in the subject language, but rather to library calls in the transformation language. Provided that the transformation language has a sufficiently powerful transformation library for the subject language, writing a transformation library extension for a domain-specific aspect is an easy task. We will demonstrate this technique by example, through the construction of *Alert*, a small error-handling DSAL extension to the Tiny Imperative Language (TIL).

The main contributions of this article are: A discussion of how the *library + notation* method for DSLs can be applied to DSALs, if the library is implemented in a meta-language; an example of the convenience of employing a program transformation language in the implementation of DSALs, compared to implementation in a general-purpose language; and a discussion of our experience with this technique for several different subject languages and aspect domains.

The paper is organised as follows. We will begin by briefly introducing our DSAL example and the TIL language (Section 8.3), before we discuss the implementation of our DSAL using program transformation (Section **??**). Finally, we discuss our experiences and related work (Section 8.5), then offer some concluding remarks (Section 8.6).

8.3 The Alert DSAL

Handling errors and exceptional circumstances is an important, yet tedious part of programming. Modern languages offer little linguistic support beyond the notion of exceptions, and this language feature does not deal with the various forms of cross-cutting concerns found in the handling of errors, namely that the choice of how and where errors are handled is spread out through the code (with `if`s and `try/catch` blocks at every corner), leading to a tangling of normal code and error-handling code. Also, the choice of how to handle errors is dependent on the mechanism by which a function reports errors—checking return codes is different from catching exceptions, even though both may be used to signal errors. Confusingly, even the default action taken on error depends on the error reporting mechanism, from ignoring it (for return codes and error flags) to aborting the program (exceptions).

The Alert DSAL allows each function in a program to declare its *alert mechanisms*—how it reports errors and other exceptional situations that arise, and allows callers to specify how alerts should be handled (the *handling policy*), independent of

the alert mechanism. We use the word *alert* for any kind of exceptional circumstance a function may wish to report; this includes errors, but may also be other out-of-band information, such as progress reports. Typical examples of alert mechanisms are exceptions, special return values (commonly `0` or `-1`) or global error flags (`errno` in C and POSIX, for instance). Ways of handling alerts include substituting a default value for the alerting function's return code; logging and continuing; executing recovery code; propagating the alert up the call stack; aborting the program, or simply ignoring the alert.

The alert extension is a good example of a domain-specific aspect language. It allows separation of several concerns: the mechanism (how an alert is reported) is separated from the policy (how it is handled), and code dealing with alerts is separated from code dealing with normal circumstances. The granularity of the policies (i.e., to what parts of the code they apply) can be specified at different scoping levels, from expressions and blocks to whole classes and packages.

Separating normality and exceptionality has already been demonstrated with AspectJ [LL00], but the AspectJ solution is less notationally elegant, and fails to separate mechanism from policy (it only deals with exceptions).[1] Using domain-specific syntax makes the extension easier to deal with for programmers unfamiliar with the full complexity of general aspect languages. Our alert extension is described in full in [BDHK06]. Here, we will look at the implementation of a simplified version for the Tiny Imperative Language.

8.3.1 The TIL Language

The Tiny Imperative Language (TIL) is a simple imperative programming language used for educational [BKVV05] and comparison purposes in the program transformation community. The grammar for TIL is given in the appendix (Section 8.7). A TIL program consists of a list of function definitions followed by a main program. TIL statements include the usual `if`, `while`, `for` and block control statements, variable declarations and assignments. Expressions include boolean, string and integer literals, variables, operator calls and function calls. We will use the name TIL+Alert for the extended TIL language.

8.3.2 Alert Declarations and Handlers

An *alert declaration* specifies a function's alert mechanisms. Our simple extension allows two ways of reporting alerts; via a condition which is checked before a call, or via a condition checked after a call. The pre-checks allow a function to report invalid

[1] We are not experts on aspect orientation, but we believe that the full separation of concerns available with our alert system is difficult if not impossible to achieve with existing general aspect languages.

Alert declaration. *Alert declarations are given after the regular function declaration. Actual arguments and the function's return value are available in the alert condition expressions. Pre-alerts have a condition that is checked before a call to the function and typically involve checks on the arguments; post-alerts are checked after the call has returned, and typically involve the return code (accessible as the special variable* `value`*, legal only in alert conditions and handlers.).*

```
FunDecl AlertDecl      -> FunDecl
"pre" Exp "alert" Id   -> AlertDecl
"post" Exp "alert" Id  -> AlertDecl
"value"                -> Exp
```

Figure 8.1: Grammar for TIL function declarations with alert extension.

parameters (before the call, avoiding the need for checks within the function itself), while the post-checks can be used for testing return values. The syntax for alert declarations is given in Figure 8.1. As an example, the following function definition declares that the function `lookup` raises the alert `Failed` if the return value is an empty string:

```
fun lookup(key : string) : string
  post value == "" alert Failed
begin ... end
```

The following declaration specifies that a `ParameterError` occurs if `f` is called with an argument less than zero, and that if the return value is `-1`, an `Aborted` alert was raised:

```
fun f(x : int) : int
  pre x < 0 alert ParameterError
  post value == -1 alert Aborted
```

A *handler declaration* specifies what action is to be taken if a given alert is raised in a function matched by its call pattern (the syntax is shown in Figure 8.2). The call pattern can be either `*` (all functions) or a list of named functions, possibly with parameter lists. This corresponds to the *pointcut* concept in AspectJ [KHH+01]. The handler itself is a statement; it can reference the actual arguments of the call (if a formal parameter list is provided in the handler declaration), names from the scope to which it applies, and `value`—the return value of the function for which the handler was called. For example, this handler declaration specifies that the program should abort with an error message in case of a fatal error:

```
on FatalError in * begin
```

Handlers. *A handler associates a statement with an alert condition; the statement is executed if the alert occurs. The* `use` *statement substitutes a value for the return value of the alerting function.*

```
"on" Id "in" {CallPattern ","} Stat -> Stat
"use" Exp ";"                       -> Stat
```

Call patterns. *A* `*` *matches a call to any function. The second form matches a call to a named function; the third form makes the actual arguments of the call available to the handler.*

```
"*"                  -> CallPattern
Id                   -> CallPattern
Id "(" {Id ","}* ")"  -> CallPattern
```

Figure 8.2: Grammar for handler declarations. The notation `{X Y}*` means `X` repeated zero or more times, separated by `Y`s.

```
  print("Fatal Error!");
  exit(1);
end
```

The `use` statement is used to "return" a value from the handler; this value will be given to the original caller as if it was returned directly from the function called:

```
on Failed in lookup(k) begin
  log("lookup failed: ", k);
  use "Unknown";
end
```

The `on`-declaration is a statement, and applies to all calls matching the call pattern within the same lexical scope. If more than one handler may apply for a given alert, the most specific one closest in scoping applies.

TIL+Alert does not add anything that can not be expressed in TIL itself, at the cost of less notational convenience. For example, given the above alert and handler declarations, a call

```
print(lookup("foo"));
```

would need to implemented somewhat like

```
var t : string;
t = lookup("foo");
if t == "" then t = "Unknown"; end
print(t);
```

This cumbersome pattern should be familiar to many programmers (programming with Unix system calls, for instance, or with C in general): save the result in a temporary variable, test it, handle any error, resume normal operations if no error was detected or if the error was handled. Exceptions alleviate the need to check for errors on every return, but writing `try/catch` blocks everywhere a handler is needed is still cumbersome, and changing handling policies for large portions of code is tedious and error-prone.

8.4 Implementation of TIL+Alert

We have several possibilities when faced with the task of implementing a DSAL, or a language extension in general:

1. Compile to object code—write an entirely new compiler for the extended language.
2. Compile to unextended language—write an aspect-weaving preprocessor for an existing compiler.
3. Compile to aspect language—write a preprocessor for an existing aspect weaver.

The first choice is typically the most costly, and therefore also the least attractive. The second option is a common technique for bootstrapping new languages, and was used for both C++ and AspectJ. The third option is only possible if the subject language we are extending already supports a form of aspects which can be suitably used for writing implementing (most of) the semantics of our DSAL. We will discuss this option in more detail in Section 8.5.

DSALs are almost by definition extensions of existing languages, and we can therefore expect to have at least some language infrastructure. In other words, we need only consider the latter two situations above. In our experience, implementing the aspect extension as library + notation in a program transformation system is a very efficient approach in terms of development time.

8.4.1 DSAL = library + notation

We have said that (alert handling) aspects are meta-programs, then showed the programmer notation for these in Section 8.3.2 where we discussed the alert grammar. This covers the "notation" half of our equation. Now we will discuss how the semantics are implemented as a transformation library written in a program transformation system.

Stratego/XT [BKVV06] is our implementation vehicle of choice. Stratego is a domain-specific language for program transformation based on the paradigm of

strategic programming [LVV03] and provides many convenient language abstractions for our problem domain. The language is bundled with XT, a set of reusable transformation components and generators—in particular a formalism for defining language syntax, called SDF [Vis97]—that support the development of language processing tools. In Section 8.5 we will discuss some of the benefits and drawbacks of using program transformation systems for implementing aspect weavers.

An existing language infrastructure for TIL exists that provides a grammar, a rudimentary compiler that does type checking and optimization, and finally a runtime that executes the compiled result. Together, these components make out a general-purpose transformation library for TIL. Using it, we can implement any program analysis and transformations on TIL programs [BKVV05]. The Alert grammar is implemented as a separate grammar module of about 30 lines of SDF code. Compositing this with the basic TIL grammar results in the complete syntax for the TIL+Alert language, c.f. the first step in Figure 8.3. We then use the TIL transformation library to implement a new Alert transformation library. Based on this, we can run meta-programs which perform the semantics of the alert constructs, i.e. the `on` and `pre`/`post` declarations: At compile-time, an abstract syntax tree for TIL+Alert is constructed and the corresponding meta-program for each alert construct is executed. Once all alert constructs in the program have been handled, the base program will have been rewritten. This completes the aspect weaving.

Ideologically, our approach can be considered an example of the "transformations for abstractions"-philosophy described by Visser [Vis05a] – we are effectively extending the open TIL infrastructure with transformations (our meta-programs) that provide new abstractions (the alerts). Next, we will describe the principles behind the implementation of the alert extension, and pay particular attention to the weaving done by the meta programs.

8.4.2 Type Checking

The constructs of the Alert language (`pre`, `post`, `on` and `use`) require their own type checking. To do this, we exploit the construction of the basic TIL type checker. It is a rule set. By adding new type checking rules to this set, we can easily extend its domain (i.e. the ASTs it can process), as we do here for `use`. The following is a Stratego rewrite rule:

```
TypecheckUse: Use(e) -> Use(e){t}
where  <typecheck-exp ; typeof> e => t
```

This rule, named `TypecheckUse`, says that if we are at a `Use` node in the AST with one subnode called `e` (this happens to be an expression), then we reuse the `typecheck-exp` function from the TIL library and annotate the `Use` node with the computed type `t`.

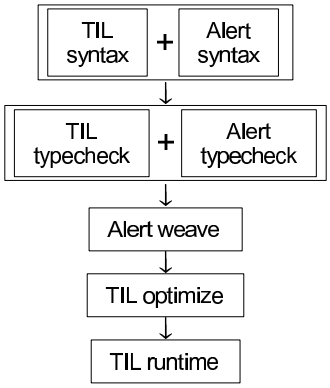

Figure 8.3: Implementation schematics. The *Alert weave* step implements the interpreter for the Alert meta-language and transforms a TIL+Alert program into a valid TIL program.

The ; operator works as function composition. The cases for `pre` and `post` are very similar. For type checking purposes, we define an `on` declaration to be a statement, thus having the `void` type. These few rules implement the "Alert typecheck" box in Figure 8.3.

8.4.3 Alert Weaving

The compilation flow in Figure 8.3 shows that after type checking, the DSAL meta-program parts of a TIL+Alert program are executed, effectuating the weaving. Once weaved, the Alert constructs are gone and the rest of the pipeline will process a pure TIL program. This program is optimised and compiled using unmodified steps of the TIL compiler.

The DSAL notation can be expanded using the simple translation scheme for we DSLs, described in the introduction, i.e. basic macro expansion, but with one crucial difference: whereas the DSL notation is expanded to library calls of a *subject* language library, the DSAL notation is expanded to library calls of a *transformation* language library, and the transformation language is generally different from the subject language. Here, TIL is our subject language and Stratego is our transformation language. Essentially, the DSAL notation is a syntactic abstraction over the Alert transformation library. This notation is embedded in the subject language (TIL), providing a distilled form of meta-programming inside TIL for managing the error handling concern.

When weaving Alert, we have to consider three constructs: the modified function definitions which now have `pre`/`post` conditions, the `on` handler declarations, and function calls. The code for the following cases are all part of the Alert transformation library where they are are implemented as Stratego rewrite rules. When the DSAL

notation is expanded, it results in calls to these rules.

Pre/Post Conditions on Function Definitions Pre/post conditions are easy to process. They are merely markers, or annotations, on the functions. The expression of a pre/post condition can only be activated by an `on`-handler, so the meta-program processing the pre/post conditions has two tasks: first, to store the alert declaration for later use, and second, to remove it from the AST so that we may eventually reach a pure TIL AST. The following rewrite rule, `WeaveFunDef`, does this:

```
WeaveFunDef:
FunDef(x@FunDeclAlert(fd@FunDecl(n, _, _), _), body) ->
  FunDef(fd, body)
where rules( Functions: n -> x )
```

It takes a function definition (a `FunDef` node) that has a subnode which is a pre/post condition (a `FunDeclAlert`) and rewrites the `FunDef` node to a pure TIL `FunDef` by removing the `FunDeclAlert` node. Further, `WeaveFunDef` creates a new, dynamic rule called `Functions` that records a mapping from the name of this function to its complete pre/post alert declaration. A dynamic rule works exactly like a rewrite rule, but can be introduced at runtime, much like closures in functional programming languages. This is done with the `rules` construct. After `WeaveFunDef` has finished, the pre/post condition is removed, and the `Functions` rule can now be used as a mapping function from the name of a TIL+Alert function to its declaration.

In the code above, _ is the wildcard pattern (matches anything) and $v@p(x)$ means bind the variable v to the AST matched by the pattern $p(x)$.

On The processing of `on` itself is also easy. Its node is removed from the AST and we add it to the current set of active `on`-handlers, maintained in the dynamic rule `On`. `On` maps from the name of an alert to the call patterns and handler for it.

```
WeaveOn: On(n, patterns, handler) -> None
where rules(On : n -> (patterns, handler))
```

Function Calls Rewriting function calls to adhere to the new semantics is the crux of the Alert DSAL, and is done by `WeaveFunCall`. This rule implements the following translation scheme. Consider the pattern for functions f in the following form, where f is the function name, f_i are the variable names, t_i are the corresponding types, tr is the return type, and the precondition is as explained earlier:

```
fun f(f0 : t0, ...) : tr
  pre exp alert signal
begin ... end
```

Whenever we see the declaration of an `on` handler, we need to process the subsequent calls in the same (static) scope, since these may now need to be transformed. We are looking for patterns on the form:

```
on signal in pattern handler;
...
z := f(e0, ...);
```

When we encounter an instance of this pattern, we may need to replace the call to f by some extra logic that performs the precondition check and, if necessary, executes the relevant `on`-handler according to the following call template[2].

```
z := begin
  var r : tr;
  var a_0 : t_0 := e_0; ...
  if exp then handler
  else r := f(a_0, ...) end
  return r;
end
```

`WeaveFunCall` will perform the aspect weaving. We will now describe the principles behind it, but not present the full source code, as this is available in the downloadable source code for TIL+Alert (see Section 8.6).

The weaving of `WeaveFunCall` can only happen at `FunCall` nodes, i.e. nodes in the TIL+Alert AST that are function calls. Assume `WeaveFunCall` is applied to a function call of the function f. First, it will check that f signals alerts by consulting the `Function` dynamic rule that was produced by `WeaveFunDef`. If indeed f has a declared alert, then the set of active `on` handlers for the current (static) scope is checked by consulting the `On` dynamic rule that was initialized by `WeaveOn`. Multiple `on` handlers can be active, so another Alert library function is used to resolve which takes precedence (the closest, most specific). Once the appropriate handler is found, the function call to f is rewritten according to the call template shown above, i.e. the FunCall node is replaced by an expression block (an `EBlock`) which does the precondition check before the call.

Extra care must be taken in the handling of variable names during this rewrite. The precondition expression is formulated in terms of the formal variable names of f, so we cannot insert that subtree unchanged. We must remap the variables, and this is done by a function called `remap-vars`. As the call template shows, for each formal parameter f_i of f, we create a local variable a_i that is assigned the actual value from

[2]The `begin/end` block here is called an expression block. It is effectively a closure that must always end in a `return`. It will be removed by a later translation step that lifts out the variables contained within it, finally giving a valid TIL program.

the call site. We rename the variables in the precondition expression of f, from f_i to a_i, and insert the rewritten expression as exp in the call template.

8.4.4 Coordination

The meta-programs induced by the `on`, `pre` and `post` declarations are dispatched by a high-level strategy that can be likened to an interpreter for the Alert aspect extension. This strategy is implemented as a traversal over the TIL+Alert AST. It contains the logic responsible for translating the Alert notation into calls to the Alert transformation library, and in that capacity, it corresponds to the DSL macro expander. Its execution will coordinate the meta-programs for the various alert constructs. Once the traversal completes, all the Alert-specific nodes will have been excised from the tree, and the result is a woven TIL AST that can be optimized and run.

8.5 Discussion

While DSLs can often be implemented as rather simple macro expanders, the same translation scheme is apparently not applicable for DSALs. The cross-cutting nature of DSALs means that statements or declarations in a DSAL usually have non-local effects. A single line in the DSAL may bring about changes to every other line in the program, and this is not possible to achieve using macro expanders. However, the translation scheme offered by the macro expansion technique is appealing both because of its simplicity and its familiarity; we already have ample experience and tools which may be brought to bear if we could reformulate the DSAL implementation problem to be a DSL implementation problem. This is what our technique offers, by using a program transformation system to implement the library (semantics) for the DSAL notation (syntax). Here, we perform a brief evaluation of our approach.

8.5.1 Program Transformation

Program transformation languages are domain-specific languages for manipulating program trees. Stratego and other transformation language such as TXL [CHHP91] and ASF [vdBHKO02] all have abstract syntax trees as built-in data types, rewrite rules with structural pattern matching to perform tree modification, concrete syntax support and libraries with generic transformation functions. The advantage to using such languages for program transformation is that the transformation programs generally become smaller and more declarative when compared to implementations in general-purpose languages, be they imperative, object-oriented or functional.

High-level Transformations In our experience, when doing experiments with aspect language and aspect weaving, working on high-level program representation

such as the AST is often preferable to lower-level representions traditionally found in compiler-backends. The AST provides all the information from the original source code and is together with a symbol table a convenient and familiar data structure to work with. When working with ASTs, it is important for the transformation language to have good support for both reading and manipulating trees and tree-like data structures.

Generic Tree Traversals Many program transformation languages and functional languages, especially members of the ML family, have linguistic support for pattern matching on trees. We have already seen pattern matching in Stratego in the rewrite rules in Section 8.4. Using recursive functions and pattern matching, tree traversals are relatively simple to express, e.g.:

```
fun visit(Or(e, e)) = ..
  | visit(And(e, e)) = ..
```

In object-oriented (OO) languages, the Visitor pattern is a common idiom for tree traversal, but compared to pattern matching with recursion, it is very verbose. Both techniques perform poorly when the AST changes, however. Introducing a new AST node type requires changes to all recursive visitor functions, or in the OO case to the interface of the Visitor (and thus all classes implementing it). There is, however, an aspect-oriented solution to the cross-cutting-concern part of this problem [**?**].

Generic programming [LVV03] in functional languages and generic traversals, as offered in Stratego, provide a solution. Generic traversals also allow arbitrary composition of traversal strategies.

```
bottomup(s) = all(bottomup(s)); s
```

This defines `bottomup` (post-order traversal) of a transformation `s` as "first, apply `bottomup(s)` recursively to all children of the current node, then apply the transformation `s` to the result". Once defined, this function can be used to succinctly program the variable renaming needed by the `WeaveFunDef` in Section 8.4.3:

```
remap-vars(|varmap) =
  bottomup(try(\ Var(n) -> Var(<lookup> (n, varmap)) \))
```

Syntax Analysis Support Program transformation languages typically come with parsing toolkits and libraries for manipulating existing languages, reducing the effort needed to create a language infrastructure. Also, there is often a tight integration between the parser and the transformation language in transformation systems. Among other things, this allows expressing manipulations of code fragments from the subject language very precisely, using concrete syntax.

Rewriting with Concrete Syntax Another important task is tree manipulation. Rewrite rules provide a concise syntax and semantics for tree rewriting, but rewriting on ASTs can of course be expressed in any language. In program transformation languages, rewriting with concrete syntax, i.e. using code fragments written in the subject language is often provided, and this may improve the readability of rewrite rules considerably, e.g.:

```
Optimize: |[ if 0 then ~e0 else ~e1 end ]| -> |[ ~e1 ]|
```

Here, `~e0` and `~e1` are a meta-variables, i.e. variables in the transformation language (Stratego) and not the subject language (TIL).

Generic Transformation Libraries Libraries for language processing are not unique to program transformation systems, but transformation libraries often contain quite extensive collections of tree traversal and rule set evaluation strategies not found elsewhere. Also, some transformation systems provide generic, reusable functionality for data- and control-flow analysis, as well as basic support for variable renaming and type analysis. However, the libraries of transformation systems are often less complete than that of general purpose languages, when it comes to typical abstract data types.

Maturity and Learning Curve A clear disadvantage of contemporary program transformation systems is their relative immaturity when compared to implementations of mainstream, general-purpose languages. The compilers are usually slower, the development environments are not as advanced, and fewer options for debugging and profiling exist. Further, the same domain abstractions that make domain-specific transformation languages effective to use, also make them more difficult to learn, a tradeoff that must be evaluated when considering the use of a transformation language.

8.5.2 Program Transformation Languages for Aspect Implementation

The stance we take in this paper is that a aspect languages are a form of domain-specific transformation language; they provide convenient abstractions (join points, pointcuts, advice) for performing certain kinds of transformations (aspect weaving—dealing with cross-cutting concerns). They hide the full complexity of program transformation from programmers. Domain-specific aspect languages are even more domain-specific, and hide the complexities of general aspects from their users.

As domain-specific transformation languages, DSALs are conveniently implemented as libraries in a program transformation language. We make this claim based

on our experience with the DSAL = library+notation method from constructing the following systems:

- A domain-specific error-handling aspect language [BDHK06]—a simplified version of this is used as an example in this paper. Our current implementation is for C, and is implemented in the Stratego program transformation language [BKVV06] using the C Transformers framework [BDD06].
- A component and aspect language for adaptation and reuse of Java classes. An early version of this is described in [BBK+05]; it is implemented by translation to AspectJ [KHH+01], using Stratego.
- AspectStratego [KV05]—an aspect-language extension to the Stratego program transformation language; implemented in Stratego itself, by compilation to primitive Stratego code.
- CodeBoost [BKHV03]—a transformation system for C++ that provides *user-defined rules*; an aspect language that allows users to declare library-specific optimization patterns inside the C++ code. The patterns are simple rewrite rules, executed at compile-time. User-defined rules is implemented with the library+notation technique, with the library written in Stratego.

Part of the design goals for many of these experiments was harnessing the expressive power of general program transformation systems into "domain-specific transformation languages" that the programmers of the subject languages could benefit from. In a word, these domain-specific transformation languages are DSALs. For most of our systems, the transformations underlying these extensions, i.e. the implementation of the DSAL semantics, are reusable Stratego libraries, and form the basis for further extensions and experiments.

Experiences One lesson learned from the construction of these DSALs is that good infrastructure for syntax extensions of the subject language is important. Reusing frontends from existing compilers usually precludes extending the syntax, as that would require massive changes to the frontend itself (and for mainstream languages, this is a substantial task). Implementing robust grammars for complicated languages like C++ and Java is infeasible, so language infrastructures provided by program transformation systems were of great help to us. Another lesson is that familiarity with language construction is crucial. Extending a subject language with an arbitrary DSAL may be very complicated, depending on what the DSAL is supposed to achieve. It may therefore be premature to expect regular developers to be able to design their own DSAL language extensions. This is often in more due to the complex semantics of the subject language itself, than the complexity of the DSAL.

8.5.3 Related Work

JTS, the Jakarta Tool Suite [BLS98] is a toolkit for developing domain-specific languages. It consists of *Jak*, a DSL-extension to Java for implementing program transformation, and *Bali*, a tool for composing grammars. Jak allows syntax trees and tree fragments to be written in concrete syntax within a Java program, and provides abstractions for traversal and modification of syntax trees. Bali generates grammar specifications for a lexer and parser and class hierarchies for tree nodes, with constructor, editing and unparsing methods. Bali supports composition of grammars from multiple DSLs. DSL development with JTS is much like what we have described here; an existing language is extended with domain-specific syntax (in Bali), and a small tool is written (in Jak), translating the DSL to the base language.

XAspects [SLS03] is a system for developing DSALs. It provides a plug-in architecture supporting the use of multiple DSALs within the same program. Declarations belonging to each DSAL are marked syntactically, picked up by the XAspects compiler and delivered to the plug-ins. The plug-ins then perform any necessary modification to the visible program interface (declared classes and methods). Bytecode is then generated by the AspectJ compiler; the plug-ins then have an opportunity to perform cross-cutting analysis and generating AspectJ code which is woven by the AspectJ compiler. Thus, implementation of a new DSAL is reduced to creating a plug-in which performs the necessary analyses and generates AspectJ code. Our method, with program transformation, can either complement XAspects, as a way of implementing XAspects plug-ins, or replace it, by developing a libraries for AspectJ manipulation in a program transformation language. The plug-in architecture of XAspects is appealing, as it forces possibly conflicting DSALs to conform to a common framework, making composition of DSALs easier. Both XAspects and our implementation can be seen as library+notation approaches. However, since domain-specific aspects in XAspects can only modify existing code using AspectJ advice and intertype declarations, there are limits to the invasiveness of the DSAL expressed with XAspects. Our implementation strategy has no such constraint since Stratego supports any kind of code modification.

The AspectBench Compiler [AAC+05] provides another open-ended aspect compiler, but is more focused on general aspect languages. It implements the AspectJ language, but is also intended as research platform for experimenting with aspect language extensions generally.

Logic meta programming (LMP) is proposed as a framework for implementing DSALs in [BMV02], because expressing cross-cutting concerns using logic languages is appealing. We believe that our approach could be instantiated with an LMP system as well: the DSAL notation may be desugared into small logic meta-programs which perform the actual weaving. Depending on the logic language, constructing and compositing logic-based transformation libraries may be possible.

In [CBE+00], the authors argue that AOP is a general discipline that should be confine itself in a domain-specific language, but rather be addressed with a general, open framework for composing all kinds of aspects. Such an infrastructure, should it be constructed, would be an interesting compilation target to expand DSAL notation to.

Gray and Roychoudhury [GR04] describe the implementation of a general aspect language for Object Pascal using the DMS program transformation system. They conclude that since transformation systems often provide good and reusable language infrastructure for various subject languages, they are good starting points when developing new aspect extensions. We are of the same opinion, and advocate a disciplined approach where the aspect extensions themselves are implemented as reusable transformation libraries that may in turn be used a substrate for later extensions.

Assman and Ludwig [AL00] describe the implementation of aspect weaving using graph rewrite systems. The authors express the weaving steps in terms of graph rewrite rules, similar to how we describe them as tree rewrite rules. In principle, transformation libraries could be constructed from the sets of graph rewriting rules, but the rule set appears to always be evaluated exhaustively. This makes rule set composition (i.e. library extension) problematic, since two rule sets that are known to terminate may no longer terminate when composed. In Stratego, there is no fixed normalization strategy; the transformation programmer may select one from the library or compose one herself, which in practice adds a very useful degree of flexibility.

8.6 Conclusion

In this paper, we have discussed the *library+notation* method for implementing DSLs: building a library that implements the semantics of the domain, a syntax definition for the desired notation, and a simple translator that expands the notation into library calls. We showed how this method can also be used effectively for implementing DSALs by writing the library part in a program transformation system, expressing the notation as a syntax extension to a subject language, and translating the notation of the DSAL into library calls in the transformation system. This makes the DSAL a meta-program that is executed at compile-time, and that will rewrite the subject program according to the implemented DSAL semantics. Our illustrating examples were based around a small imperative language with an aspect extension for separately declaring error handling policies.

We argued that, based on our experience, program transformation systems are ideal vehicles for implementing such libraries because they themselves come with domain-specific languages and tools for doing language processing, which greatly reduces the burden of implementation when compared to general purpose languages.

The complete implementation of TIL+Alert is available at `www.codeboost.org/alert/til`.

8.7 TIL Grammar

Programs. A program is a list of function definitions, followed by a main program (a list of statements).

```
FunDef* Stat* -> Program
```

Functions. A function definition defines is function with a given signature (`FunDecl`) and body (a list of statements).

```
"fun" Id "(" {Param ","}* ")" ":" Type -> FunDecl
FunDecl "begin" Stat* "end"            -> FunDef
Id ":" Type                            -> Param
```

Statements:

```
"var" Id ";"                               -> Stat
"var" Id ":" Type ";"                      -> Stat
Id ":=" Exp ";"                            -> Stat
"begin" Stat* "end"                        -> Stat
"if" Exp "then" Stat* "end"                -> Stat
"if" Exp "then" Stat* "else" Stat* "end"   -> Stat
"while" Exp "do" Stat* "end"               -> Stat
"for" Id ":=" Exp "to" Exp "do" Stat* "end" -> Stat
Id "(" {Exp ","}* ")" ";"                  -> Stat
"return" Exp ";"                           -> Stat
```

Expressions:

```
"true" | "false"      -> Exp
Id                    -> Exp
Int                   -> Exp
String                -> Exp
Exp Op Exp            -> Exp
"(" Exp ")"           -> Exp
Id "(" {Exp ","}* ")" -> Exp
```

Lexical syntax:

```
[A-Za-z][A-Za-z0-9]*    -> Id
[0-9]+                  -> Int
"\"" StrChar* "\""      -> String
~[\"\\\n] | [\\][\"\\n] -> StrChar
```