Creating and using domain-specific language features*

Position Paper

Ted Kaminski
Department of Computer Science and Engineering
University of Minnesota
Minneapolis, Minnesota
tedinski@cs.umn.edu

Eric Van Wyk
Department of Computer Science and Engineering
University of Minnesota
Minneapolis, Minnesota
evw@cs.umn.edu

ABSTRACT

The value that domain-specific languages provide to their users is the domain-specific language features they contain. These features provide notations from the domain of interest, as well as domain-specific analysis and optimizations. But domain-specific *languages* are sometimes a poor means of delivering these valuable features to their users. A challenge arises when a problem crosses multiple domains and whose programming or modeling solution could benefit from language features from all domains of interest. Using multiple domain-specific languages can become cumbersome, perhaps outweighing their benefits in the first place.

An alternative approach, advocated by this position paper, is to provide domain-specific language *features* to programmers and modelers as composable language *extensions* that they can import into their general-purpose programming or modeling language. In our view, there are three requirements for a language extension framework to be widely usable. First, language extensions should be developed independently, by domain-experts, as libraries or domain-specific languages are now. Second, extensions should be automatically composable so that programmers and modelers can pick the language extensions they want, and direct tools to compose them, without the need for writing "glue-code." Third, this composition process should not fail to yield a *working* compiler (or other tools) for the custom extended language. Thus, the programmer has some assurance that the extensions that they pick will work together.

We briefly describe how this vision of extensible language frameworks is supported by the Silver and Copper metaprogramming tools.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Extensible languages

*This work is partially supported by NSF Awards No. 0905581 and 1047961.

GlobalDSL '13, Montpellier, France

1. INTRODUCTION

Domain-specific languages [2] (DSLs) provide language features tailored to a particular area of interest in the form of concrete syntax that incorporates notations from the domain and semantic analysis and optimizations that are typically not easily achieved in general purpose languages. Challenges arise when a programming or modeling problem crosses multiple domains and one chooses to use multiple DSLs. Using multiple DSL tools, which precipitates the use of multiple files for programs and models in different languages, is only manageable on a small scale. As the number of DSLs being used grows this integration problem becomes more tedious.

Domain-specific languages, even when used individually, also pose some challenges. Maintenance challenges can result in useful DSLs such as LN [5], which has optimizations for computational geometry problems, to be no longer supported and thus difficult to use. Some DSLs, such the YACC parser generator, are translated to a general purpose language and allow one to write phrases in this language, but they perform no syntactic or semantic analysis on these embedded phrases; errors are only determined when the generated code is compiled. A similar problem occurs when DSL phrases are written as strings in a general-purpose language, as the JDBC library does for SQL queries. Syntactic and semantic errors are not determined at compile time.

Programmers and modelers use DSLs because of the features that they provide. But a complete, independent *language* is often a poor mechanism for delivering these features. Our hypothesis is that the problems described above can be overcome if the domain-specific features in these languages are provided to users not as a new language but as *composable language extensions* that can be imported into a general-purpose programming or modeling language. Extensible language frameworks should allow programmers, who know little about language design and implementation, to import domain-specific language features into a host general purpose programming or (textual) modeling language.

A slightly contrived example of a program using a collection of language extensions might help to clarify. In Figure 1 is a program written in a version of Java that has been extended with three domain-specific language extensions. The first adds concrete syntax and type checking for SQL queries. The program uses new features to establish a connection, `c`, to a database and then uses that connection to `import` the schemas of two tables, `sites` and `depots`. Next a new feature is used to write an SQL query that extracts the latitude and longitude of tourist sites that are classified as historic, as

```
public class Demo {
 void demo ( ) {
  connection c = "jdbc:/db/geo.db" ;
  import table sites from c ;
     /* FLOAT lat, FLOAT lon, VARCHAR category */
     /* Latitude, longitude, and category of
        tourist sites  */
  import table depots from c ;
     /* FLOAT lat, FLOAT lon, STRING rentalCar */
     /* Train depots latitude, longitude, and
        availability of a rental car */

  ResultSet geomData = using c query {
    SELECT sites.lat, sites.lon,
           depots.lat, depots.lat,
           depots.rentalCar
     FROM  sites, depots
     WHERE sites.category = "Historic" } ;
```

```
  while ( geomData.next() ) {
    UBRational s_lat = geomData.getFloat(1) ;
    UBRational s_lon = geomData.getFloat(2) ;
    UBRational d_lat = geomData.getFloat(3) ;
    UBRational d_lon = geomData.getFloat(4) ;
    Boolean d_rental = geomData.getString(5) ;

    Boolean walkableDistance =
      sign ( ... s_lat, s_lon, d_lat, d_lon, 1.5 ... ) ;
    Boolean driveableDistance =
      sign ( ... s_lat, s_lon, d_lat, d_lon, 75.0 ... ) ;
    Boolean canVisit = table {
      walkableDistance  : T * ,
      driveableDistance : * T ,
      d_rental == "Y"   : * T } ;
  }
 }
}
```

Figure 1: An example program written in using composable language extensions.

well as train depots and rental car availability at the station. By using this SQL language extension, the extended compiler is able to parse the query to detect syntax errors and perform semantic analysis to detect type errors. It translates the query, like all extension constructs, down to plain Java code, in this case code that uses the JDBC library.

In the second column, the program extracts latitude and longitude into an unbounded-precision rational number type introduced by a second extension that incorporates features from LN [5],a computational geometry DSL. The new `sign` construct determines if a value computed over these rational types is positive. Here an expression, which is elided, determines if the tourist site is within a circle of radius either 1.5 or 75.0 whose origin is at the depot. This is a common primitive performed over geometric data. The extension restricts expressions so that it can statically determine the number of bits needed to represent the unbounded precision value. This allows many optimizations to be applied; *e.g.*, loops that operate over these unbounded precision values can be statically unrolled. These optimizations result in dramatic speed-ups in computing these sorts of primitives [5].

Finally a extension for writing Boolean-valued expressions in a tabular form is used to determine if the site can be visited. This `table` extension consists of rows of Java expressions (which might contain other nested language extensions) followed by an equal number of flags indicating if the value should be true (`T`), false (`F`), or either (`*`). Here, the site can be visited if `walkableDistance` is true or if both `drivableDistance` and `d_rental == "Y"` are true.

To make language extension practical, it must be possible for programmers or modelers, who are not experts in language design and implementation, to simply select the language extensions that they want to use, while the language extension framework takes care of the rest. We believe three characteristics are required to realize this.

1. To ensure that a rich collection of extensions are available they must be independently developed by different parties; one extension writer creating, for example, the SQL extension above, and another creating the extension for computational geometry.

2. The programmer or modeler should not need to understand the underlying language implementation techniques and thus the extensions should be automatically composable. Tools carry out the composition of the host language and extensions without the need to write any "glue code" to compose them.

3. Furthermore, this composition must always be successful; despite the fact that even though many forms of declarative specifications such as context free grammars naturally compose, the composition is not always one from which a compiler or translator for the extended language can be generated (for example, the grammar may be ambiguous).

These lead to two challenges: first, composing specifications for concrete syntax so that a scanner and parser can be generated, and second, composing specifications of semantic analyses so that error checking and optimization of the various extension constructs can be performed.

2. COMPOSABLE CONCRETE SYNTAX

Two immediate problems in composing concrete syntax jump to mind. First, how do we cope with new keywords introduced by different language extensions? In the previous example, two extensions introduce `table` as a keyword and this must be recognized as a different (terminal) symbol in different contexts. Second, how can we be sure that the automatically composed context free grammar, from which a parser will be generated, is non-ambiguous?

Copper [17] is an integrated parser and context-aware scanner generator that addresses these, and other, issues. The declarative specifications used by Copper are familiar in form, but are interpreted in a slightly different way. A

parser is constructed from the context-free grammars specified by the host language and language extensions, and a scanner is generated from regular expressions associated with each terminal symbol in the grammar.

From these specifications Copper generates an LALR(1) parser that is slightly modified, in that it provides contextual information to the generated scanner so that it can differentiate terminal symbols that have overlapping regular expressions, such as the two terminal symbols for the keyword `table`. When a Copper-generated parser calls its scanner for the next token, it passes the scanner all terminal symbols that are valid in the current state: those whose entry in the LR-parse table for the current LR-parse state is either *shift*, *reduce*, or *accept*, but not *error*. The scanner will then only return a token for a terminal in this set.

In the fourth line of the example program, after the parser has shifted the `import` token, it is in an LR-state in which the terminal symbol matching `table` defined in the SQL extension is *valid* but the terminal symbol defined in the tabular-Boolean-expression extension that also matches `table` is not. Thus the scanner will return only the appropriate (valid) token. This ability to use context to distinguish between terminal symbols with overlapping regular expressions is crucial for parsing composed languages using a deterministic (LR) parser. We have found that the flexibility provided by a context-aware scanner can significantly reduce the sense of brittleness sometimes associated with LR-parsers. Because terminal symbols do not need to be overloaded for different uses in different context, a more natural grammar can be written that more easily fits into the LALR(1) class.

But to address the second problem of grammar ambiguity, we use a *modular* analysis that is performed by the individual language extension developers. It checks that their extension, when combined with other independently-developed extensions that also pass this analysis, will form a context free grammar from which a deterministic LALR(1) parser can be generated. The guarantees provided by this *modular determinism analysis* [10] can be state precisely as follows:

$$(\forall i \in [1,n].\ \mathit{conflictFree}(CFG^H \cup CFG_i^E) \wedge \mathit{isComposable}(CFG^H, CFG_i^E)\ ) \implies \mathit{conflictFree}(CFG^H \cup \{CFG_1^E, \ldots, CFG_n^E\})$$

This states that if each individual extension's context free grammar (CFG_i^E) when combined with the common host language grammar (CFG^H) is free of conflicts in its LR-parse table ($\mathit{conflictFree}(CFG^H \cup CFG_i^E)$) and satisfies additional structural requirements of the grammar and derived constructs such as follow-sets ($\mathit{isComposable}(CFG^H, CFG_i^E)$), then when any combination of extensions that pass this analysis are combined with the host language, the composed grammar is also free of conflicts, and thus non-ambiguous.

This analysis satisfies the second and third requirements (the second and third enumerated points on the previous page), but, as one would expect, at the cost of putting some restrictions on the syntactic extensions that can be made. One of these restrictions requires an initial keyword "marking" token at the beginning of the right hand side of any extension production with a host language production on the left hand side, for example, the `using` and `table` keywords in the example above.

One might consider using generalized parsing techniques such as SGLR [13] and GLL [11] instead. These automatically compose and thus satisfy our second requirement. However, because there is no assurance that the composed grammar is not ambiguous they fail to satisfy our third requirement. A parser that returns multiple parse trees, as generalized parsers may do, does not "just work" since the non-expert programmer must determine which tree is the correct one. This requires the type of language implementation knowledge that general programmers do not have. Another option is projectional editors, such as those in Intentional Programming [12] and MPS [18]; these alleviate the need for a textual parser but do require a system-specific editor.

The full details of this analysis, along with a discussion of related work, can be found in previous papers by Schwerdfeger and Van Wyk [17, 10]. The second paper argues that the imposed restrictions are not unreasonable and it also describes an analysis that ensures there will be no lexical ambiguities encountered by the scanner.

3. COMPOSABLE SEMANTIC ANALYSIS

Silver [14] is our extensible attribute grammar system that is used to specify the semantics of the host language and language extensions. Like context free grammars, attribute grammars (AGs) naturally compose. Extensions can introduce new non-terminals, productions, attributes, etc. as well as equations for new attributes for existing host language productions. A mechanism also exists to add new contributions to existing equations so that, for example, an extension can perform a new semantic analysis and add additional error messages into an existing list-of-errors attribute for existing host language productions.

A key feature for composition in Silver is forwarding [15]. During attribute evaluation a production can define a semantically equivalent tree from which it will automatically get values for attributes that do not have defining equations. Inherited attributes are automatically copied to the "forwards-to" tree from the "forwarding" tree. This enables a solution to a form of the "expression problem"; if one extension defines new attributes for host language productions and another adds new productions, the new productions can forward to a host language tree for values of attributes defined in the other extension. For example, the tabular expression extension above forwards to a plain Java expression of nested conjunction (`&&`) and disjunction (`||`) expressions.

Even though AGs naturally compose, the composition is not always complete or *well-defined*; that is, the composition may be missing equations that define the value of certain attributes introduced in the different extensions. But Silver has a *modular well-definedness analysis* [7], similar in form to the one in Copper, that ensures that the composition of AGs that pass this analysis, called *modComplete*, will be well-defined. This is stated precisely below:

$$(\forall i \in [1,n].\mathit{modComplete}(AG^H, AG_i^E)) \implies \mathit{complete}(AG^H \cup \{AG_1^E, ..., AG_n^E\}).$$

If a language extension (*e.g.* AG_i^E) does not pass this analysis then the extension writer has the opportunity to fix the problem. This analysis performs structural checks on the grammar determines what inherited attributes are needed to compute each synthesized attribute and imposes some restrictions on these to ensure that the composition of grammars will be well-defined. Our previous paper [7] on this analysis provides the full details and a more complete discussion of related work than is possible here.

4. CONCLUSIONS

We have used SILVER and COPPER to build extensible language frameworks for both programming and textual modeling languages. ABLEJ is our implementation of Java 1.4 in which some Java 5 features were added as languages extensions. These include the enhanced-for-loop and auto-boxing and unboxing of primitive types [16]. These techniques also work for textual modeling languages as demonstrated by the SILVER specifications of Lustre [6], a synchronous language, and Promela [9], the modeling language for the SPIN model checker. Modeling languages often lack linguistic support for some commonly-used general purpose and less commonly-used domain-specific concepts; this leaves the engineer to encode them as idioms, a time-consuming and error-prone process that can be avoided when appropriate language extensions are used. Both modular analyses restrict what extension writers can specify as *composable* language extensions, but we found the restrictions quite reasonable given the strong guarantees that they provide; our past work [10, 7] provides the details of these restrictions.

There is much other work in extensible languages using other implementation techniques. To mention just a few, there is the SugarJ [4] extensible Java implementation is based term rewriting and scannerless generalized parsing in the Spoofax framework [8], the Neverlang[1] system based on a notion of "roles," and the extensible JastAdd Java compiler [3] based on reference attribute grammars. These approaches target a slightly different audience that we have, however. They support the re-use and extension of declarative language specifications but lack the modular analyses in SILVER and COPPER that provide strong guarantees that the automatic composition of specifications will result in a translator that will "just work." Consequently, their audience is primarily those who have some compiler implementation expertise.

Currently we are using SILVER and COPPER and their modular analyses to develop an extensible specification of C into which we are adding extensions targeting the domains of data-mining and high performance computing. One challenge is just in the design and implementation of these extensions and their included optimizations so that efficient parallel code can be generated. Another challenge is to design the host language so that it is as extensible as possible. Different host language designs (and implementations) can be more easily extended in different ways. We thus see "extensibility" as another (host) language design criterion.

5. REFERENCES

[1] W. Cazzola. Domain-specific languages in few steps: The Neverlang approach. In *Proc. of Intl. Conf. on Software Composition (SC)*, volume 7306 of *LNCS*, pages 162–177. Springer, 2012.

[2] A. v. Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[3] T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *Proc. of OOPSLA*, pages 1–18. ACM, 2007.

[4] S. Erdweg, T. Rendel, C. K astner, and K. Ostermann. SugarJ: Library-based syntactic language extensibility. In *Proc. of OOPSLA*. ACM, 2011.

[5] S. Fortune and C. J. van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Trans. on Graphics*, 15(3):223–248, 1996.

[6] J. Gao, M. Heimdahl, and E. Van Wyk. Flexible and extensible notations for modeling languages. In *Fundamental Approaches to Software Engineering, FASE 2007*, volume 4422 of *LNCS*, pages 102–116. Springer-Verlag, March 2007.

[7] T. Kaminski and E. Van Wyk. Modular well-definedness analysis for attribute grammars. In *Proc. of Intl. Conf. on Software Language Engineering (SLE)*, volume 7745 of *LNCS*, pages 352–371. Springer-Verlag, September 2012.

[8] L. C. L. Kats and E. Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In *Proc. of OOPSLA*, pages 444–463. ACM, 2010.

[9] Y. Mali and E. Van Wyk. Building extensible specifications and implementations of Promela with AbleP. In *Proc. of Intl. SPIN Workshop on Model Checking of Software*, volume 6823 of *LNCS*, pages 108–125. Springer-Verlag, July 2011.

[10] A. Schwerdfeger and E. Van Wyk. Verifiable composition of deterministic grammars. In *Proc. of Conf. on Programming Language Design and Implementation (PLDI)*, pages 199–210. ACM, June 2009.

[11] E. Scott and A. Johnstone. GLL parsing. *Electronic Notes in Theoretical Computer Science*, 235:177–189, 2010.

[12] C. Simonyi, M. Christerson, and S. Clifford. Intentional software. *SIGPLAN Not.*, 41(10):451–464, 2006.

[13] M. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In *Proc. 11th International Conf. on Compiler Construction*, volume 2304 of *LNCS*, pages 143–158, 2002.

[14] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: an extensible attribute grammar system. *Science of Computer Programming*, 75(1–2):39–54, January 2010.

[15] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *11th Conf. on Compiler Construction (CC)*, volume 2304 of *LNCS*, pages 128–142. Springer-Verlag, 2002.

[16] E. Van Wyk, L. Krishnan, A. Schwerdfeger, and D. Bodin. Attribute grammar-based language extensions for Java. In *Proc. of European Conf. on Object Oriented Prog. (ECOOP)*, volume 4609 of *LNCS*, pages 575–599. Springer-Verlag, 2007.

[17] E. Van Wyk and A. Schwerdfeger. Context-aware scanning for parsing extensible languages. In *Intl. Conf. on Generative Programming and Component Engineering, (GPCE)*, pages 63–72. ACM, 2007.

[18] M. Voelter. Language and IDE development, modularization and composition with MPS. In *GTTSE 2011*, volume 7680 of *LNCS*, pages 383–430. Springer, 2011.