ECOOP 2004 Workshop Report: Evolution and Reuse of Language Specifications for DSLs (ERLS)

Organized By
Thomas Cleenewerck[1]
Krzysztof Czarnecki[2]
Jörg Striegnitz[3]
Markus Völter

Edited By
Thomas Cleenewerck[1]

[1] PROG, Vrije Universiteit Brussel, Belgium
[2] University of Waterloo, Canada
[3] Research Centre Juelich, Germany

Abstract. This report summarizes the results of the workshop on evolution and reuse for language specifications for DSLs. The focus of the workshop was twofold: exploration of the current research activities concerning reuse and evolution of language specifications and discussion of the identification, extraction, and composition of reusable parts of DSL specifications. The workshop combined presentations with focused discussions on these emergent topics: reusable assets, role of object-orientation, conflicts among reused assets, and the quest for a DSL test-case example to facilitate and guide future discussions.

1 Introduction

With the advent of rewrite rule systems, template transformation languages and attribute grammars, the development of compilers for domain-specific languages has been greatly facilitated. Despite these technologies compiler development is usually started from scratch, i.e. the grammar and the semantic specifications are completely written starting from nothing. Unfortunately the cost of adapting an existing DSL implementation or the implementation of another similar DSL often boils down to the same amount of work. Naturally, given the already existing implementation of a DSL, this cost is too high.

The aim of this workshop was to bring researchers together and provide a forum to discuss the issues in DSL development and evolution with the particular focus on identifying, extracting, and composing reusable parts of DSL specifications. We specially concentrated on, but not limited the workshop to the use of object oriented techniques and concepts like encapsulation and inheritance to make DSLs more reusable.

Since this was the first workshop of its kind, the focus of the workshop included a wide range of topics concerning evolution and reusability to explore current research activities. The topics of interest include but were not limited to:

- Implementation technologies and paradigms: transformation systems, meta-programming, interpreters
- Reusable assets ranging from individual transformations to patterns
- Reusability techniques: customization, configuration, adaptation, encapsulation, compositions and inheritance

Of course not all of the above suggested topics could be discussed. The following discussion topics are the topics that emerged during the workshop in the afternoon: reusable assets, role of object-orientation, conflicts among reused assets, and the quest for a DSL test-case example to facilitate and guide future discussions.

The workshop report consists of summaries of the contributions in section 2 and reports of the discussions and the results of the topics 3. Table 1 lists all the participants and their affiliation. The position papers are available at the workshop website (`http://prog.vub.ac.be/~thomas/ERLS/`).

2 Contributions

The contributions are ordered as follows: two cases, a formalization and two implementations. We start off with two contributions where a domain is presented and for DSLs where there is a great potential for reuse and evolution and the conflicts that may arise during reuse. The two domains are respectively user interfaces and middleware patterns. The implementation of the former is based on logic rules and the implementation of the latter by a transformation system with various extensions. The next contribution investigates and attempts to formalize object oriented approaches using descriptive logics for capturing transformational knowledge. The following two contributions each incorporated reuse in their transformation system, respectively ReRags and TLG.

2.1 The Write Once, Deploy N MDA Case Study Combining Performance Tuning with Vendor Independence (Pieter Van Gorp)

Summary In this paper, we presented a complex middleware pattern as a realistic case study for model-driven development. From this case study we derived concepts of a transformation language that would allow us to generate such patterns towards different application servers. The language supports the specialization of common transformations, the separation of concerns, and evolution conflict resolution. For the latter feature we applied the familiar concept of design by contract. Following this principle, code generation can be integrated with

architectural consistency rule checking: consistency constraints can be considered as the postconditions of transformation rules. Failed postconditions trigger a reconciliation unit that can be considered as the body of a transformation rule. Reconciliation units can often be generated from declarative OCL postconditions. For new components, reconciliation will result in code generation. Evolution conflicts in existing software require reconciliation units that correct the inconsistencies without regenerating other items. For this purpose, such units can analyze source and target models by navigating traceability models that maintain a persistent link between these two. Since conflict resolution logic cannot always be generated by the transformation engine, transformation specifiers need language support to implement it manually. By carefully inheriting from common superclasses in the design of metamodels for PIMs and PSMs, one can specify consistency constraints across abstraction layers in a very concise manner. At a low level of abstraction, imperative code templates can be integrated with the model (to model) refinement process.

2.2 A Declarative DSL Approach to UI Specification - Making UI's Programming Language Independent (S. Goderis, D. Deridder)

Summary A large number of researchers are trying to find innovative ways to tackle the problem of separation of concerns. In our research we focus on the concern of User Interfaces and application interactions because these are seldom separated from each other. A clean separation would nevertheless facilitate UI development and evolution. Figure 1 depicts this idea of separating the UI from its underlying application. One should keep in mind that both interact externally with each other and internally with themselves. When separating concerns it will therefore be important to focus on factoring out these interactions since they are the main cause for the code entanglement. In our work we focus on the interactions between the UI and the application (figure 1 number 3), and within the UI itself (figure 1 number 2), as well on the different layers of abstraction within the GUI box (figure 1 number 1).

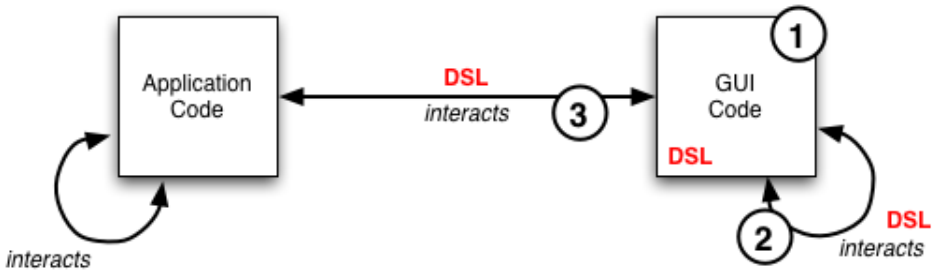

Fig. 1. Separating GUI and Application Interactions

Once the separation between UI and application is achieved, different examples can be thought of where separation is beneficial. For instance, deploying the UI to another platform without having to rebuild completely, or adapting the interface to different kinds of users.

In order to solve this problem we will express high level UI components (i.e. the GUI components in figure 1 (1 and 2)) by means of declarative meta-programming. This denotes that a user interface is described declaratively by means of rules and facts and later on is composed through a problem solver. Different concerns, as well as different UI abstraction layers, will lead to different sets of rules. The problem solver will combine all of these in order to get to the user interface that is wished for. Consequently we provide a declarative user interface framework as a domain specific language to program user interface concerns and their interactions with the application code.

As shown in figure 1 we can consider the declarative specifications for the User Interface and the interactions as being a DSL, namely for the domain of User Interfaces.

Relation to the workshop We have encountered several issues related with DSL's for which the workshop could provide us with new insights.

Inheritance Reusing existing rule-bases can be done by parameterizing one rule-base by another one. We need to investigate how to ensure inheritance between different UI sets, or how to make them dependent on each other. This is also related to evolution issues.

Customization and Configuration We want to generate the UI based on the declarative rules and facts. Nevertheless we should not restrict ourselves to static User Interfaces since a lot of the UI power is currently left unattended because of this staticness. It should be possible to customize a User Interface dynamically. For instance, when someone resizes a window, another layout strategy might have to be applied. These layout strategies should be applied to the user interface dynamically, without need for recompiling the static UI.

Another issue related to this topic is putting constraints between the different UI building blocks [EL88,FB89]. For instance, how to specify that a certain component should be placed above another one (and what to do if conflicts arise)?

Evolving and Adapting It is not clear which abstractions will be needed. It is not possible to anticipate all possible abstract UI components, as this will depend on the developers' point of view as well. Furthermore a certain abstraction might play a role in different UI framework parts, and it is not always clear where to draw the line. We acknowledge that it will never be possible to completely separate the UI from the application (since they have to interact at some point in time). But also on the level of the application interactivity layer it is not clear where to stop making abstractions.

Once the basic entities are defined, they are composed into more complex and abstract entities. A user will want to use these higher level entities without having to care about lower level entity descriptions. This implies that users will consider the high level entities as black boxes without knowing what the low level entities actually look like. Therefore two high level entities may contradict each other without the user knowing it.

2.3 Knowledge Representation in Domain Specific Languages (D. Hutchins)

Summary Most domain-specific languages are layered on top of an existing general-purpose language, such as C or Java. As a result, it is convenient to think of individual DSLs (or DSL fragments) as program transformation systems; they translate domain-specific constructs into a base language.

Transformation systems can be difficult to use because they operate on abstract syntax trees, and do not have any real understanding of the *meaning* of the code they modify. Two different transformations may conflict if they alter the code base in incompatible ways. I argue that capturing the semantic meaning of domain-specific concepts is the real challenge of developing reusable DSLs.

A domain-specific language encodes knowledge about the problem domain. Instead of encoding knowledge by translating domain-specific concepts into C or Java, I propose to define such concepts directly, using a dedicated knowledge-representation language. Under this model, a DSL or DSL fragment is represented as an ontology of concepts and relations between concepts. Combining two different DSLs together is a matter of merging the two ontologies.

Description logics (DLs) are a knowledge representation system that is currently used by the artificial intelligence community. DLs as they are ordinarily defined are essentially a type system – they can describe the interface of objects, but they cannot be used to implement the behavior of objects. This limitation prevents them from being used to implement DSLs.

In this paper, I introduce the Sym calculus, which is a formal object calculus that extends descriptions logics into a full-fledged programming language. This language not only supports standard functions, classes, and generics, but it can be used to implement virtual types, mixins, features, components, and aspects. Sym achieves this flexibility because it is based on a knowledge representation system, and can thus handle the semantics of terms at a much more abstract level than mainstream OO languages.

Relation to the workshop The focus of this workshop was using object-oriented techniques to make DSLs more reusable. Most proposals discussed ways of using OO constructs to implement program transformation systems.

It is well known that one way to implement a grammar is to define each non-terminal as a class. If a non-terminal has more than one definition then it can be defined as an abstract base class, with each variation as a concrete subclass. The grammar thus becomes a class hierarchy. This is a particularly

attractive mechanism for implementing more advanced systems, such as attribute grammars, because the class hierarchy can be used to annotate the abstract syntax tree with additional information beyond that what's present in the raw syntax.

The advantage of using classes to represent non-terminals is that each node in the abstract syntax tree has a type and a well-defined interface. This means that the AST can be queried and manipulated via method calls, just like any other OO data structure.

This is essentially the philosophy that I have taken with the Sym calculus. My approach is to ignore the question of syntax altogether, and focus on developing more flexible ways of defining and manipulating complex object structures.

Concepts in Sym are defined much like OO classes, but they may be related to other classes via type equations. Type equations are evaluated at compile-time. This means that Sym can be used as an "extensible compiler" architecture for implementing DSLs. If an abstract syntax tree is constructed at compile time (via an external parser), then information will propagate through the tree at compile-time. Unlike the OO method-call mechanism, there is no run-time penalty to using domain-specific constructs.

2.4 Reusable Language Specification Modules in JastAdd II (T. Ekman, G. Hedin)

Summary This paper discusses how to build domain specific languages (DSL) on top of a general purpose language (GPL) using our language implementation tool JastAdd II. The specification formalism in JastAdd II is based on Rewritable Reference Attributed Grammars (ReRAGs) [EH04], that combines object-oriented abstract grammars, static aspect-oriented programming, reference attributed grammars, and conditional rewriting.

The technique is illustrated by evolving a matrix framework into new language constructs that are added on top of Java. The GPL, in this case Java, is extended at both the syntactic level and the semantic level. The semantic extensions affect name analysis as well as type analysis. The code generation finally transforms the extensions into the base language by using a matrix computations framework. The extension is done in a modular way and re-uses large part of the static semantic analysis from the base grammar.

Relation to the workshop The DSL approach used in the paper is based on the JastAdd II tool, a combined attribute grammar and transformation system. The paper focuses on how to express DSL extensions and GPLs in a modular fashion and how to compose new languages from these individual parts. The same technique can be used to customize an exisiting language.

The specification formalism in JastAdd II allows modular specification and provides several synergistic mechanisms for separation of concerns, e.g. inheritance for model modularisation, static AOP for cross-cutting concerns, rewrites that allow computations to be expressed on the most suitable model, and declarative formalism to allow transparent composition of attributes and rewrites. This

enables the DSL and GPL specifications to be expressed separately in modular fashion and combined in a transparent way.

2.5 Object-Oriented Language Specifications: Current Status and Future Trends (M. Mernik, X. Wu, B. R. Bryant)

Summary The challenge in domain-specific language definition is to support modularity and abstraction in a manner that supports reusability and extensibility [MM03]. The language designer wants to include new language features incrementally as the programming language evolves. Ideally, a language designer would like to build a language simply by reusing different language definition modules (language components), such as modules for expressions, declarations, etc. These reusable components should be straightforwardly extendible to reflect language design changes.

Our position statement is that the use of object-oriented techniques and concepts, like encapsulation and inheritance, improves language specification languages to a much greater extent towards their modularity, reusability and extensibility than any other technique. In this paper two of our latest approaches are briefly described. In the LISA approach the attribute grammar as a whole is subject to inheritance employing "Attribute grammar = Class" paradigm. We call this multiple attribute grammar inheritance. With our approach, the language designer is able to add new features (syntax constructs and/or semantics) to the language in a simple manner by extending lexical, syntax and semantic specifications. In the second approach, we combine Object-Oriented Two-Level Grammar [BL02] specifications with Java to implement domain-specific languages. We specify the lexical, syntax and abstract (domain-independent) semantics rules by TLG specification and implement the concrete semantics by user-supplied Java classes. We also apply several object-oriented design patterns (interpreter pattern, composite pattern, chain of responsibility pattern) to help improve the modularity and abstraction level of TLG specification to enhance the reusability of formal specifications in language implementation.

Relation to the workshop Object orientation plays an important role in the design of language specifications for DSLs. This paper is a position paper that describes the evolution of object-oriented language specifications. It explores the current status of applying modular formal specification in language definitions and introduces several modern language specification approaches, such as object-oriented attribute grammar [Paa95], intentional programming [dM01], two-level grammar [van74], action semantics [DM03], JTS[BLS98], JJForester [KV01], etc. Current shortcomings and future trends of language specification language are also introduced at the end of this paper.

We conclude in the paper that the use of object-oriented techniques and concepts, like encapsulation and inheritance, improves language specification languages to a much greater extent towards their modularity, reusability and extensibility than any other technique. To illustrate this, our two latest object-oriented approaches are introduced in the paper. The attribute grammars we

used in LISA system have been modeled as objects, which is similar to Jast-Add II introduced by Ekman and Hedin. Likewise the two level grammars in our second approach are also modeled and designed after objects. We go even further and state that many object oriented design patterns are in fact useful for the implementation of a TLG-based compiler generation system. Our two approaches have been proven successful in developing various domain-specific languages (e.g. SODL, COOL, AspectCOOL, PLM, PAM, Matrix language). Our experience with these non-trivial examples shows that object-oriented specification languages are very useful in managing the complexity, reusability and extensibility of language definitions. Specifications become much easier to read, maintain and to modify.

3 Discussions

3.1 Reuse

Reuse of DSL specifications is hard, because of their domain dependent nature. Lifting part of these specifications to general language constructs and generally applicable semantics is therefore not easy. We compiled a non-exhaustive list of general constructs and semantics that are most likely to be found in many DSL implementations:

Value construction The computation of new values and new AST nodes.

Value propagation and distribution The propagation of new values throughout the tree and the distribution of the values to the AST nodes that require these values. Various propagation schemes like tree traversals, xpaths, attribute inheritance and synthesis, etc., already exist.

Name resolution and binding In the grammar names are merely a syntactic entity represented by some kind of an identifier or hierarchical identifier. Because the actual semantic meaning of names is often context dependent, a name resolution or analysis process must resolve their semantic meaning.

Type analysis Statically typing (whether it is explicit or implicit in the DSL language) is a cornerstone to insure the correctness of a program written in a DSL. In order to infer type correctness type analysis is required which (1) connects the use sites to the declaration sites and (2) checks the operations in the use sites against the operation supported by the type.

Construction of typed nodes according to a pattern The type of an AST node plays in many transformation systems a crucial role because it is one of the quantifiers to select certain transformations and computations on the tree. However, the exact type can often only be determined after some analysis of the tree. Therefore during transformation, the obtained information is used to construct new AST nodes of a more appropriate type.

Resolving non-local results of one-to-many transformations One-to-many transformations exert an influence in the form of non-local AST nodes on various parts of the parse tree. Appropriate mechanisms are needed to deal with such non-local nodes.

Null-value elimination Structural mismatches between the source and target language break the fundamental in-place substitution principle of many transformation systems because certain source AST nodes cannot be substituted by the produced new AST nodes after their transformation. When this is the case, null-values or empty subtrees must be removed from the tree.

The items in this list are closely related to implementation, compiler and language infrastructure. How broad or narrow the notion of infrastructural mechanism can be interpreted was a point of discussion. Some argued that DSL editors, etc., were also part of this infrastructure, while others rather narrowed this notion to implementation mechanisms. An item that is particularly difficult to categorize are semantic validators. If the semantics of a language is described via a transformation to another language, then semantic validators would not be considered as infrastructure. When the semantics of a DSL is expressed with some kind of formalism, validators based on that formalism would be considered as infrastructure. Currently the reuse of language modules is largely investigated in context of evolving a DSL to another version. Out of the discussions we strongly believe that there is a lot reuse potential for language modules belonging the same domain. More research and field expertise is needed to further broaden and increase reuse for the development of DSLs. The exploration of reuse among researchers that are often not familiar with the domain proved to be very difficult. Concrete cases of reuse could help us understand the required mechanisms and the current difficulties better. Since reuse is rather at its early stages, these cases are currently not available.

3.2 Role of object orientation

Object orientation plays an important role in the design of todays transformation systems. The introduction of OO techniques and concepts in transformation systems facilitates the development of new languages, increases the modularity of the components and allows more reuse. Let us discuss each of the techniques and concepts borrowed and used of OO in the design of a transformation system.

The notion of an object as an encapsulated entity with state and behavior gets applied to language modules (either BNF rules, rewrite rules, linglets, templates or attributes depending on the kind of transformation system). Language modules become configurable entities containing data and behavior. In Delesley, language entities are concepts modeled after objects with fields and functions. The attribute grammars discussed by Ekman and Bryant have been modeled as objects where productions correspond to classes and attributes to methods. Likewise the two level grammars discussed in Bryant are also modeled and designed after objects. Finally VanGorp points out that template transformation languages like xDoclets -used to implement his model to code transformations- are encapsulated entities containing datamembers and functions. Inheritance is the mechanism object orientation offers for code reuse and code specialization. In class based languages subclasses can override and inherit behavior and data, respectively specialize and reuse existing code. Inheritance has been successfully

adopted towards language modules of DSLs. In the JastAdd II transformation system described in the paper of Ekman, name analysis and typing for a matrix extension to Java is introduced with a minimal amount of effort hereby reusing existing name analysis and specializing typing rules. VanGorp argues that there is a lot of potential for an inheritance mechanism to reduce the code base containing the model to model and model to code transformations of MDA for distributed enterprise Java applications. Bryant goes even further and shows that many object oriented design patterns are in fact useful for the implementation of a TLG-based transformation system.

Besides inheritance as a reuse mechanism there are other mechanisms as well. Goderis is exploring how to achieve reuse in the context of logic programming by specializing certain rules or providing extra alternatives. This way existing rules are reused and new behavior is accented for.

Because of this encapsulation, DSL designers can now pay more attention to modularity, in terms of the relationship of the language components with the rest of the compiler. Modularity is important because it is one of the corner stones for reusability. In other words, encapsulation brought us a step closer to our goal.

3.3 Conflicts

The reuse of language constructs and semantics can be hampered by conflicts and information dependencies that occur when two or more of those reusable elements are combined. The definition of a conflict was a point of much debate in the workshop. From the debate the following preliminary definition of a conflict can be summarized as follows: a conflict is a non-trivial resolution of two language components, which have a mutual interest. An interest of a language component can be every part of the language on which the component depends:

Constrains the conditions under which a component may be executed
Requirements the conditions which a component relies on when executing
Inputs the information or structures the component needs to performs its task
Outputs the information or structures the component produces and/or changes
of a conflict categorization of conflicts

In target-driven transformation systems the conflicts are resolved manually in the modules that produce the targets. These modules actually deal with several concerns at the same time. The transformations are in this regard not well modularized from a separation of concerns viewpoint. Source-driven transformation systems tried to unhook most of the different concerns and put them in separate modules. As a result of this separation of concerns, the conflicts that were manually tackled must now be tackled by the transformation systems itself. There exist various schemes to cope with conflicts. The most common approach is to define an order by scheduling the two language components to achieve the desired combined result. The prerequisite for this schema is that there must exists some correct partial order in which the language modules must be executed.

But for language modules which both create the same entity with a different content, ordering is not sufficient and more invasive composition mechanisms are needed like the ones supported in LTS (Linglet Transformation System). Additionally the constraints and requirements of language modules could also be more taken into account. The semantics of a DSL is often specified by its translation to another language. Although this is a formal semantics, it is hard to derive any properties form such a description. Up till now, one could consider typing as the most widely spread implementation of this idea. A richer mechanism like operational semantics, descriptive logic, f-logic, ontology semantics, etc., would certainly help to detect and even resolve conflicts. Despite most of these formalisms being around for quite some time, very few DSL designers seem to revert to them. There are several reasons for that but the main reason is that it is for most DSLs (like for example specification languages) quite unclear what we would like to describe. Krzysztof suggested the use of state charts as an alternative mechanism. Although it doesn't capture the full semantics, there are interesting properties that can be calculated out of a start chart. In an offline discussion Bryant and Cleenewerck argued that a more suitable and specific semantic formalism is needed. Some of the attendees pointed out that a lot of those conflict resolution schemes can be found in other research tracks like aspects, hyperspaces, subject oriented programming, etc. The ideas and mechanisms of these schemes probably could contribute to the existing transformation systems.

3.4 DSL Test-case example

The discussions were often broken off or stalled because of the absence of a single shared test-case DSL. The problems mentioned were initially very abstract because we didn't want to delve to deep in a particular problem domain and implementation approach. Soon we realized that such statements are no help for anybody. Not every participant had the same background making it impossible to understand the statement. In order to concretize some of the statements, the statements were rephrased with more problem domain information. Although this helped a bit, since most of the participants were not familiar with problem domain itself, the general problem domain knowledge was difficult to grasp. Moreover, in a single afternoon there is not enough time to discuss the problem domain, the general concepts and the ideas behind the domain-specific statement. So after such an attempt it was up to the other participants to recognize the abstract statement in their problem domains. Not seldom due to the absence of such recognition of ideas and statements there was only a poor response. It became rapidly clear that a shared test-case DSL would greatly ease the discussions.

In response to that need we tried first to get a more solid grip on the true nature of a domain-specific language. In Ekman and Bryant a DSL is an set of syntactical and semantical extensions to a general purpose language. In DeLesley a DSL is described with a set of concepts and rules described using logics. The DSLs of VanGorp are meta-models which are refinements (model to model transformations) and model to code transformations. In Goderis a DSL is a logic

program querying facts to produce an effect. Clearly there is no strict definition of what a DSL is. We found that DSLs can be classified according to number of axes:

Language axis The language axis refers to the relationship between GPL (general programming language) and DSLs. DSLs range from GPL programs, to extensions of GPLs all the way to an entirely new language.

Structural axis The structural axis refers the amount of flexibility and mechanisms to compose a program. In GPL a program is a complex composition of fine-grained language constructs. Any construct can almost be composed with any other construct. The more domain-specific a DSL is, the more the composition of language constructs is restricted. At the far-end of this spectrum we find the specification languages, which constitute merely of the assigning of values to properties. Domain-specifity is thus more a matter of a degree. Because of this wide range of DSLs we may need to revert to a set of smaller DSL examples instead of searching after the holy grail in programming languages.

As a first attempt to reach a shared test-case DSL we distilled and reformulated our research questions to a set of requirements to which such a common DSL should adhere to.

A Declarative DSL Approach to UI Specification The three most important and distinguishing research questions in the context of a declarative DSL approach to UI specifications are:

- How to make keywords dependent on each other
- Which keywords should be provided and which keywords can be extended. And how to allow a developer to extend keywords.
- How to ensure a developer does not violate the meaning of a certain keyword when extending it such that contradictions are avoided.

From these research questions we can compile the following list of requirements to which a test-case DSL should adhere to:

- Dependencies among keywords
- Open variability's in the language which must be provided and/or extended.
- Conflicting and non-conflicting constraints caused by simple use of the existing keywords or by the extensions of them.

Knowledge Representation in Domain Specific Languages There are various DSL implementation techniques with very different capabilities. A formal comparison of these techniques would reveal how knowledge is captured and made executable. From that comparison, properties on the kind of information that can be handled can be derived and used to establish a hierarchy of DSL techniques and DSLs, much like the regular-language, context-free language, and Turing machine hierarchy in complexity theory.

For example: Small-step operational semantics is a classical way of formally defining programming languages. Small-step semantics is based on the process of term *reduction*. Each term is reduced to a simpler term, until it becomes a *normal form* which cannot be further reduced. Reduction has two nice properties:

(1) Reductions only make use of local information. Reducing a term only affects that particular term; it does not affect any surrounding terms. Assuming that the reduction rules are confluent, reductions can be applied in any order.

(2) User-defined reductions can be supported within a language by means of partial evaluation. If an expression only references constants, then it can be evaluated (i.e. reduced) at compile-time. This mechanism is completely transparent and requires very little in the way of compiler support; it's what the Sym calculus uses to handle type equations.

Full transformation systems which use rewrite rules are considerably more powerful than partial evaluation. A rewrite rule can affect any number of terms in any way. Rewrite rules are also especially prone to clashes and conflicts, which makes them difficult to use and debug.

The important question here is, in the context of common DSL test-case, what kinds of DSLs require the full power of a rewriting transformation system, and what kinds can be implemented with simpler processes.

Reusable Language Specification Modules in JastAdd II Our paper deals with the approach where a GPL is extended with a DSL, and where the DSL extensions are expressible in the base language. The DSL is transformed into the GPL either through direct translation to GPL code or through calls to a framework. This situation is useful when the generated code is to verbose to hand code or to complicated to write by hand and therefore needs additional semantic checks. The DSL high-level representation may also simplify domain specific optimizations.

Some questions that form the basis for our work include:

Composition How can we express the language specifications to enable transparent composition of separate GPL and DSL specificiations?

Separation of concerns How can we achive good separation of concerns in the language specifications? We would like to separate the logical phases in the compiler such as name binding, type analysis, and code generation while at the same time allowing language constructs to be expressed in a modular fashion.

GPL API How do we model a suitable API to the GPL that the DSL can use to reuse existing components such as name binding and type analysis? Can the DSL also contribute to these components, e.g. add new scopes to the name binding module?

A suitable test-case DSL for this research must require a tight integration with the target language (in this case the GPL) compilation phases, e.g. name binding, type analysis, etc., on the one hand, but remaining modular with respect to the target language and other extension module on the other hand.

Object-Oriented Language Specifications: Current Status and Future Trends As already mentioned, the use of object-oriented techniques and concepts greatly improves language specification languages towards better modularity, reusability and extensibility. To achieve modularity, extensibility and reusability to the full extent these techniques need to be combined with aspect-oriented techniques since semantic aspects also crosscut many language components. Moreover, special algorithms have to be invented (e.g. forwarding) to improve modularity of underlying formal methods. Another shortcoming of current approaches is lack of scalability since they do not fully support grammatical operators such as described in [Wil99]. The ideal solution where the language designer can freely combine language components based on different formal methods is less likely to appear in forthcoming years.

In order to experiment or validate further research in this direction a test-case DSL is required which is more like a family of DSLs or a DSL production line constructed out of a set of different language components.

4 Conclusion

Summarizing what was said in each discussion topic, we conclude that :

- There is a lot of potential for reuse in (1) languages that must be customized to support different sub-domains by means of language extensions and in (2) languages that evolve overtime to keep up with changes in their associated domains. Reusable language specifications for various domains are still rather rare and tend to be more related to language implementation mechanism then to language features.
- Object orientation plays an important role in the design of todays transformation systems. The introduction of OO techniques and concepts in transformation systems facilitates the development of new languages, increases the modularity of the components and allows more reuse.
- The reuse of language constructs and semantics can be hampered by conflicts and information dependencies that occur when two or more of those reusable elements are combined. A richer mechanism like operational semantics, descriptive logic, f-logic, ontology semantics, etc., but customizable and more suitable for DSLs would certainly help to detect and even resolve conflicts. Also the ideas and mechanisms of conflict resolution schemes available in aspects, hyperspaces, subject oriented programming, etc. probably could contribute to the existing transformation systems.
- The discussions were often broken off or stalled because of the absence of a single shared test-case DSL. As a first attempt to reach a shared test-case DSL we distilled and reformulated our research questions to a set of requirements to which such a common DSL should adhere to.

Table 1. List of workshop participants.

Name	Affiliation	E-mail Address
Thomas Cleenewerck	*Vrije Universiteit Brussels, Belgium*	tcleenew@vub.ac.be
Krzysztof Czarnecki	*University of Waterloo, Canada*	czarnecki@acm.org
Jörg Striegnitz	*Research Centre Juelich, Germany*	J.Striegnitz@fz-juelich.de
Markus Völter	*none*	voelter@acm.org
DeLesley Hutchins	*University of Edinburgh, UK*	d.s.hutchins@sms.ed.ac.uk
Barrett R. Bryant	*University of Alabama at Birmingham, USA*	bryant@cis.uab.edu
Pieter Van Gorp	*Universiteit Antwerpen, Belgium*	Pieter.VanGorp@ua.ac.be
Simon Dobson	*Trinity College, Dublin IE*	simon.dobson@cs.tcd.ie
Sofie Goderis	*Vrije Universiteit Brussels, Belgium*	sgoderis@vub.ac.be
Torbjörn Ekman	*Lund University, Sweden*	torbjorn.ekman@cs.lth.se
Görel Hedin	*Lund University, Sweden*	gorel@cs.lth.se

References

[BL02] B. Bryant and B.-S. Lee. Two-level grammar as an object-oriented requirements specification language. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)-Volume 9*, page 280. IEEE Computer Society, 2002.

[BLS98] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–153, Victoria, BC, Canada, 2–5 1998. IEEE.

[dM01] O. de Moor. Intentional programming., 2001.

[DM03] Kyung-Goo Doh and Peter D. Mosses. Composing programming languages by combining action-semantics modules. *Sci. Comput. Program.*, 47(1):3–36, 2003.

[EH04] Torbjörn Ekman and Görel Hedin. Rewritable Reference Attributed Grammars. In *Proceedings of ECOOP 2004*, volume 3086 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[EL88] E. Epstein and W.R. Lalonde. A smalltalk window system based on constraints. In *Proceedings of OOPSLA88*. ACMPress, 1988.

[FB89] B. Freeman-Benson. Constraint technologie for user-interface construction in ThingLabII. In *Proceedings of OOPSLA89*. ACMPress, 1989.

[KV01] Tobias Kuipers and Joost Visser. Object-oriented tree traversal with jjforester. In Mark van den Brand and Didier Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier, 2001.

[MM03] T. Sloane M. Mernik, J. Heering. When and how to develop domain-specific languages. Technical Report Technical Report, SEN-E0309, CWI, 2003.

[Paa95] Jukka Paakki. Attribute grammar paradigms a high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2):196–255, 1995.

[van74] A. van Wijngaarden. Revised report on the algorithmic language ALGOL 68. *Acta Inf.*, 5:1–236, 1974.

[Wil99] D. Wile. Integrating syntaxes and their associated semantics. Technical Report Technical Report, USC/Information Science Institute, 1999.