A Role-based Approach Towards Modular Language Engineering

Christian Wende[1] and Nils Thieme[1] and Steffen Zschaler[2]

[1] Institut für Software- und Multimediatechnik
Technische Universität Dresden
Dresden, Germany
c.wende@tu-dresden.de
[2] Computing Department
Lancaster University
Lancaster, UK
szschaler@acm.org

Abstract. Modularisation can reduce the effort in designing and maintaining language specifications. Existing approaches to language modularisation are typically either focused on language syntax or on language semantics. In this paper, we propose a modularisation approach covering both syntax and semantics. We propose defining composition rules on the level of abstract syntax, making it the central artefact in a language module. To enable clean interfaces for such language modules—effectively making them language components—we use role-modelling at the metamodel level.
We discuss how role-based metamodelling supports the aspectual modularisation of language semantics and can also be integrated with concrete syntax specifications to build self-contained language components. We present the implementation of our approach in the LanGems language compositions system and show how it can be used to provide a modularised definition of the Object Constraint Language.

1 Introduction

Software engineers use an increasing manifold of languages for all kinds of tasks in areas like software analysis, design, implementation, build automation, testing, or configuration [1]. If such a language is applied to a special domain, we speak of domain-specific languages (DSLs). DSLs capture the domain's particular abstraction and allow to formulate statements about it in a tailored way. The benefits of this approach are obvious and generally accepted [2].

However, the advantages come along with the additional effort to implement and maintain each language. This effort can be reduced by language modularisation in two fundamental ways. First, by enabling reuse of language modules in language development and, second, in reducing the complexity of language maintenance. Both require the components to be *self-contained*, that is syntactically and semantically complete and reusable independently of other components.

A lesson that research in component-based software engineering taught us, is that reuse of components never comes automatically, but requires a sensible design of components for reuse and a dedicated system for component composition [3, 4]. The design of language components has intensely been studied based on a modularisation of the concrete syntax of a language [5–8]. There, grammar inheritance provides means for composing existing grammar components. A purely syntactic approach does not exploit the full potential of language composition, because lots of the effort for developing languages results from implementing language semantics. Hence, a language component should contribute a syntax and semantics. There are also semantics approaches (e.g. attribute grammars [9, 10]) that allow for modularising language specifications. Unfortunately, their integration with modular parser generators requires manual synchronisation of concrete syntax components and semantics components. This is partly due to the fact that the syntax imposes a modularisation based on language constructs whereas semantics aspects (e.g., name and type resolution) crosscut the syntactic structure [9].

To address these issues, we introduce a component-based language engineering approach that makes abstract syntax the central artefact in a language component. The contributions of this paper are as follows:

1. We discuss how a role-based extension for abstract syntax metamodelling supports the aspectual modularisation of language semantics and also integrates with concrete syntax specifications to build self-contained language components.
2. We show that the explicit language component interfaces introduced by roles and the composition technique of role composition preserves independence of language components.
3. We present *LanGems*—a dedicated language composition system implementing the role-based approach.
4. We discuss benefits and problematic issues of the approach in a case study modularising the Object Constraint Language (OCL).

The next section introduces our role-based approach to component-based language engineering at a conceptual level. Section 3, then briefly discusses our prototypical implementation. In Sect. 4, we discuss our findings from applying the approach to a modularisation of OCL. Finally, Sect. 5 discusses related work and Sect. 6 concludes the paper.

2 Role-based Language Engineering

According to [11, 12] the specification of a language consists of three dimensions: a metamodel to describe the language's *abstract syntax*, a *concrete syntax* specification to describe the language's (in our case textual) syntax and a description of the language *semantics* that assigns meaning to language expression (i.e., instances of the metamodel). The objective of building self-contained language components requires all three dimensions to be considered and integrated. In the following we will extend abstract syntax metamodelling to support component-based language engineering while integrating concrete syntax and semantics.

	Inheritance	**Subtyping**
Intention	share commonalities (attributes, references)	reuse generic behaviour by anticipated variation
Information Hiding	no hiding	hiding of private information
Reuse Granularity	within components	over component borders

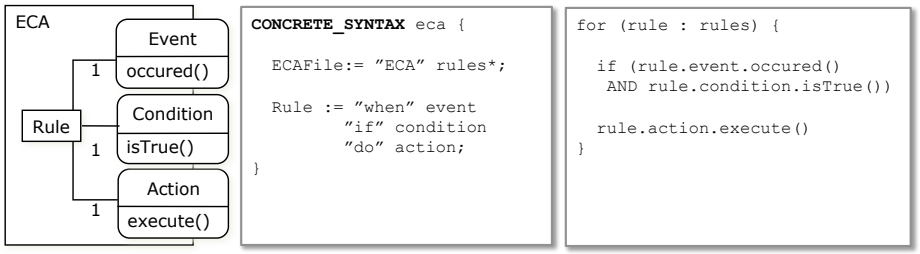
Table 1. Comparision of Subtyping and Inheritance

2.1 Extending Abstract Syntax Metamodelling by Role Modelling

The purpose of an abstract syntax metamodel is to specify the abstraction of a language by describing the structure of concepts found in its domain, the properties (attributes) of these concepts and their relationships (references). Metamodels define a graph-based data structure for representing language expressions and are more expressive [12, 13] than abstract syntax trees (ASTs) [11] historically used. Indeed the AST still remains in form of the model's spanning tree that is made up by the containment hierarchy of the model elements. The additional cross references in the model result from type and name analysis. Thus, models are considered as appropriate to represent all information that can be derived at compile time. They serve as exchange structure between a language's concrete syntaxes that are used before compile time and the language's dynamic semantics that is relevant after compile time. Their integrating position justifies the primacy of language metamodels for developing self-contained language components.

The quality of a language composition approach is determined by a clean modularisation of language components and a sensible method for component composition. Analysing common metamodelling languages [14–18] shows one main issue regarding modularisation: The common practice of using packages to build components of related metamodel elements together with direct references and subclassing to interconnect components is considered harmful: Component interconnection is expressed as part of the component specification, making components more difficult to reuse. The inheriting or referencing component needs to know the component it inherits from or references to. The components are not independent and, thus, not maintainable and usable individually. In addition, subclassing combines the concepts of *subtyping* and *inheritance* which is discussed very controversially [19, 20]. While subtyping expresses that objects of a subtype can be used wherever their supertype is expected (i.e., in references defined between the types), inheritance describes that concrete features (i.e., attributes, references and operations) defined by an (abstract) superclass are propagated to all subclasses. As concluded in Tab. 1 this mixes two intentions in metamodel design: reuse of generic behaviour by anticipated variation (subtyping) and reuse by sharing of common properties (inheritance). The combination of both in subclassing breaks the principle of information hiding between components, since inherited properties can be accessed and altered in arbitrary ways. We propose subtyping as the mechanism of choice to interconnect components while inheritance should be restricted as reuse formalism within components.

A second aspect discouraging subclassing as the sole reuse mechanism for modular language engineering is that it operates on the granularity of classes whereas language components typically consist of a set of related metamodel elements.



a) Role-based abstract syntax metamodel for the ECA component

b) Concrete syntax specification for the ECA component

c) Operational semantics specification for the ECA component

Fig. 1. Example of role-based language component

To address these issues, we propose to extend current metamodelling approaches with

1. an explicit distinction between subtyping and inheritance in metamodel design, and
2. an explicit composition technique for language components that conserves information hiding and component independence.

To do so, we transfer the idea of role modelling [21–23] to abstract syntax metamodelling. This gives us both an appropriate abstraction (terminology) to define language components and a technique for their composition. Role models introduce an orthogonal dimension in the design of object-oriented systems. They provide means to specify behavioural aspects that crosscut the static system structure. Role models do not look at individual types, but describe how behaviour (or semantics) is realised by the interaction of many types. They modularise the generics of a behavioural aspect of the system. Each participant of this interaction that is described by a role is expected to contribute a specific part to this behaviour. How this is realised is not described in the role model, but in the connection of the roles with the static system structure. Thus, roles define both placeholders for class types in the role model and a semantic contract for objects that are supposed to play the role.

The connection of role-based modelling and class-based modelling of the static system structure is specified using the played-by relation. It connects roles types with classes whose instances (objects) can play a given role. The connection does not only define a subtype relationship from the class to the role, but also how class instances fulfil the role's semantic contract. Please note, that our role-based approach only supports subtyping at class level. Subtype relations between other metamodel-elements (e.g., attributes or references) are not supported.

Figure 1a) shows how role modelling is used to define the abstract syntax of a language component. The example describes a component of an Event-Condition-Action Language (ECA) [24]. It uses a metamodelling language that was extended by an explicit distinction of role types (rounded rectangles) and class types (edged rectangles). The metamodel reads as follows: An ECA specification consists of an arbitrary number of `Rules`. `Rules` have a triggering `Event`, a `Condition` that must be satisfied and

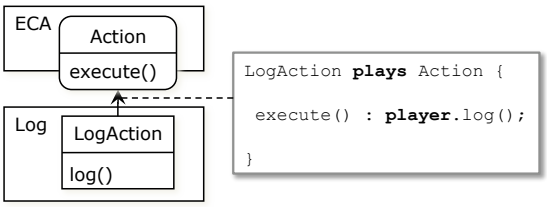


Fig. 2. Example for language composition

an `Action` that is executed. Note that these four concepts are sufficient to describe the full semantics of an ECA specification. This characteristic makes the depicted ECA metamodel self-contained. The types `Event`, `Condition`, and `Action` are represented as role types. That means their inner semantics is underspecified in the context of the ECA component. They only define a semantic contract by a number of *role operations*, that need to be realised by a concrete role-playing class type. Thus, roles define the explicit interfaces for our language modules—effectively making them language components. The ECA component also contains the class type `Rule`. It is not modeled as a role-type, since it is not expected to be externally modified during language composition.

As mentioned above, the played-by relations connect roles types with classes whose instances (objects) can play a given role. In our language composition approach we use this relation to interconnect the abstract syntax of several language components. In accordance to this connection also concrete syntax and semantics of the components are integrated. Figure 2 shows an example of connecting the ECA component with a simple language for logging. Here, `LogAction` plays the role `Action`, with the semantics of the role operation `execute` being defined by the `log` operation.

Since instances of a class that plays a role can be used wherever the role is expected, the played-by has a subtyping semantics [22]. Private properties of the role type are only accessible in the context of component and not inherited to the role-playing class. This decouples subtyping from inheritance and, thus, conserves information hiding. Note, that the role-playing relationships are not specified within the components, but externalised to a composition program (a simple example composition program is shown in Fig. 2). None of the components directly depends on a concrete other component— they are defined separately and identified by their unique name. Concluding, role interfaces serve two purposes: They keep the language components independent and define the structural and semantic constraints for the component interconnection.

2.2 Integrating Concrete Syntax Specifications

The concrete syntax of a language describes how expressions are presented to the user. The possible representations are manifold: Traditionally, we think of a textual syntax, defined using a concrete syntax specification like Extended Backus-Naur Form (EBNF) [11]. With the rise of Model-Driven Software Development (MDSD), diagrammatic syntaxes gained importance and even the tree-based model editors found in current

modelling tools (e.g. EMF [25]) can be considered as concrete syntax for language expression. All these representations have in common, that they are defined in relation to the abstract syntax of a language. Textual parsers transform a textual syntax into a model instantiating the language metamodel, diagram editors use specific graphical primitives to distinguish model elements regarding their abstract types, and tree-editors combine a graphical representation of the containment associations between model elements with a textual and form-based representation of element attributes and references.

To be integrated with the role-based specification and composition of abstract syntax concrete syntax specification faces two specific challenges:

1. The concrete syntax specification needs to correspond to the granularity of abstract syntax components, and
2. compositions of the abstract syntax models needs to be reflected during composition of concrete syntax.

And as a more general issue of concrete syntax composition

3. conflicts and ambiguities in the composed syntaxes need to be addressed.

These challenges apply for all kinds of concrete syntaxes, but this paper will primarily focus on textual languages. To achieve this, we need to bridge the gap between concrete syntax formalisms and abstract syntax metamodelling. In previous work [26] we presented EMFText—a tool that addresses this bridging problem by integrating a parser generator with the metamodelling framework EMF [25]. The paper discussed the conceptual relations between Ecore a metamodelling language for abstract syntax and CS an EBNF-like language to define concrete syntax. The introduced mapping and its realisation in EMFText has special characteristics that directly tackle the aforementioned challenges.

First, every concrete syntax rule is mapped directly to a high metaclass in the languages metamodel. With this we can guarantee that the specification of concrete syntax corresponds to the granularity of abstract syntax components. Role types introduced in role-based language engineering correspond to external interfaces of a language component. They do not contribute a concrete syntax on their own, but reuse the syntax of the types they are played by. Thus, EMFText do not need to be extended to support syntax rules for role types. In Fig. 1b) the syntax for the ECA language is described in the EBNF-like syntax used by EMFText. It only gives a concrete syntax for the class type `Rule` but leaves the syntax of the role types open. The role types act as syntactic interface decoupling the language component's concrete syntax specifications.

The composition of language components results the combination of their syntax rules in a single grammar. For each subclass relationship between two metaclasses, EMFText generates a corresponding choice rule in the textual grammar, where the left-hand side corresponds to the super-class and the right-hand side is a choice between all the subclasses. This reflects the subtyping semantics of subclassing. We extended this behaviour to played-by relations used to compose components in our role-based approach. The concrete syntax for the role types is defined as the choice between the concrete syntax of all role players defined during language composition.

For the ECA example, the grammar for `Action` would be defined as `LogAction | EmailAction | ...`.

The hardest problem in syntax composition is related to conflicts and ambiguities introduced by concrete syntax composition. These result when the integrated language components have an overlapping syntax. There are two necessary steps to tackle this issue. First, one needs to detect these overlaps. Second, the conflicts need to be solved. Most parser generation formalisms detect overlaps during the generation of a parser from the composed parser specification. There are different grammar formalisms that are more or less restricted w.r.t syntactic overlaps. For instance scannerless parsers [27, 28] allow overlapping in keywords by taking the context where the keyword is parsed into account. However, ambiguities in the composed grammar result in sets of alternative parse trees that have to be disambiguated manually [29]. To avoid manual disambiguation of parse trees, the parser generator we used in EMFText is more restrictive. It does not allow syntactic overlaps. Conflicts are detected and reported during language composition. In the case of ambiguities the language developer is requested to remove them by adapting the syntax specifications. This restrictiveness is reasonable for modularising single languages, since they should be unambiguous in themselves. In composition scenarios involving more than one DSL disambiguation relies on the freedom to adapt the DSLs concrete syntax.

2.3 Integrating Semantics

Language semantics describes how meaning is associated to language constructs. To integrate semantics in role-based language engineering it is necessary to investigate the connection of the introduced roles and played-by relationship with language semantics. The function of roles is twofold: Within the semantic specification of a component they describe behaviour provided by the participants of the collaboration. The role operations are used in defining the components semantics. During the composition of language component, role types define the interface that has to be implemented by all role players. They decouple the language components semantically and contribute a semantic contract between them. Compared to a subclassing-based composition as found in current metamodelling formalisms they provide information hiding principles for language components, but also explicit constraints during composition.

Integrating semantics in role-based modelling needs considering two issues:

1. Defining the semantics within a language component, and
2. describing the semantics of language composition.

In general, semantics defines the meaning of a language (component) in a transformation of a language expression into the *semantic domain* of the language. Literature [12, 30, 31] distinguishes several formalisms for this purpose (e.g., operational, denotational, translational, or extensional approaches). Being very different regarding their implementation, they all share a common characteristic: They operate upon the languages abstract syntax (metamodel).

Within a language component we do not differentiate between the type interface provided by role types and normal class types. Thus, the specification of semantics

within the context of components is not any different to defining semantics for metamodels. Role-based language components can be combined with existing semantics formalism that work for metamodels. An example is given in Fig. 1c). It contains an operational definition of the semantics to evaluate ECA rules: All `rules` are checked if their `event` was triggered. If this is the case the rules `condition` is checked. When this condition evaluates `true`, the corresponding `action` is executed.

Another formalism particularly suited to be combined with role-based language composition for modularising language semantics are attribute grammars as described in [32, 9]). In [33] and [34] the authors describe the realisation of a modular Java compiler using a declarative mechanism for extending the attribution of the abstract syntax tree. Single semantic aspects that crosscut the syntactic structure of the AST can be specified in individual modules. However, attribute grammars make no explicit distinction between component semantics and composition semantics. They mix both the generic semantics of a special semantic aspect (e.g. type analysis) and its binding to concrete nodes of the AST in the same module. Consequently, the semantics is modularised but still depends on the AST it is bound to. To realise self-contained and reusable language components it is vital to decouple reusable semantics of a component and semantics interconnection. A sensible design of semantics modules (as described for attribute grammars in [35]) could achieve this separation. Role-based language composition leverages this idea—it insists on this separation by differentiating between component semantics and composition semantics. Fig. 2 depicts a fragment of the composition program for connecting the ECA component with the simple logging component. Semantics composition is described in the played-by relation between the `Action` role and the `LogAction` of the logging component. For this the realisation of the `Actions` role operation `execute()` is described in the context of `LogAction`. It corresponds to the invocation of the `log()` operation. This specification describes the semantic adaptation necessary to adjust the role player to the roles required interface. The example demonstrates that the specification of composition semantics applies the same semantics formalism as the specification of component semantics. The only difference is the changed context. This contextual separation of component semantics and composition semantics ensures information hiding between composed components and preserves there semantic independence.

3 Implementation of a Role-based Language Composition System

This section will present the realisation of the role-based language engineering approach proposed in Sect. 2. We describe the fundamental constituents of our language composition system *LanGems*[3] and its model-driven process for language composition.

Generally, a composition system is a triple consisting of a *component model*, a *composition language*, and a *composition technique* [4]. The component model describes

[3] The name is derived from the terms morpheme and lexeme: During the lexical analysis of a program, a lexeme describes the smallest unit in a parser's input stream. Morphemes denote the smallest entities with a defined semantics. In analogy, at the level of language specifications the term LanGem refers to a self-contained set of concepts that realise a particular language feature.

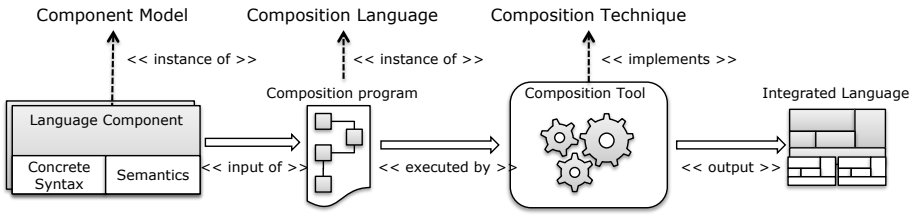


Fig. 3. Overview of the language composition process in LanGems

how components—in our case language components—look like and how they can be accessed. The composition language introduces the vocabulary used to describe concrete composition programs that specify the combination of several components to a system—in our case an integrated language. And finally, the composition technique defines the technological mechanism that actually realises the composition.

Fig. 3 depicts the central elements and coarse structure of the LanGems composition process. Individual language components are specified based on the *LanGems Component Specification Language* (LCSL) which uses the concepts of LanGems component model. Every language component specifies its concrete syntax and semantics in relation to its metamodel. The composition of several language components is specified in a composition program formulated in the *LanGems Composition Language* (LCL). This program is evaluated by a composition tool, that implements the language composition technique and generates an integrated language. It consists of a combined metamodel, a composed concrete syntax, and a composed semantics.

Component Model The central artefact of every language component is its abstract syntax metamodel. For the purpose of defining a self-contained, but extensible language component, the Language Component Specification Language (LCSL) enables role-based metamodelling by extending the metamodelling language Ecore provided by EMF [25]. Ecore provides a mature and extensible metamodelling infrastructure and its expressiveness is comparable to common metamodelling languages, e.g, EMOF [15], KM3 [14], or GOPPRR [18]. As depicted in Fig. 4 the LCSL introduces `Class`es and `Role`s as subclass of `EClass` to distinguish the two special kinds of metaclasses. Classes are provided to allow for traditional subclassing inheritance commonly used in current metamodels. Roles introduce types that are meant as target of played-by relations; that is, they are to be extended through subtyping. Additionally, they define special `RoleOperation`s behaving like conventional operations within the component. As discussed before, these role operations are also carriers of semantic contracts between components and have to be redefined with specific semantics for each role player. Thus, their second function is to contribute to the external required interface of the component relevant during composition. On the other hand, the set of `classTypes` define potential role-players for component composition and, thus, contribute the provided interface of a language component. Other concepts like attributes and references known from conventional metamodelling are inherited from `EClass`. A `Component` contains all role and class types belonging to the language component.

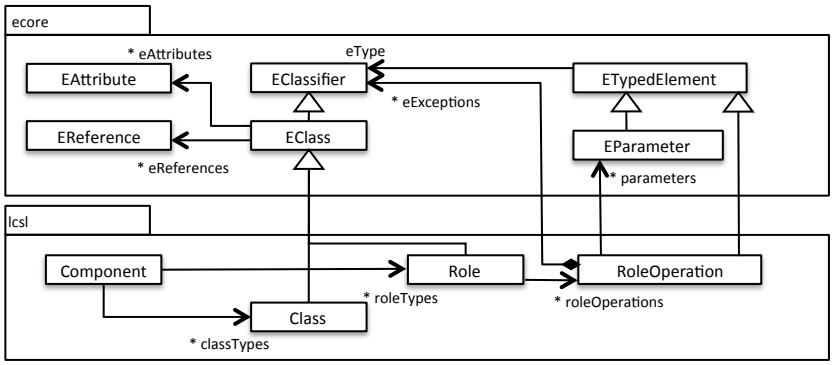


Fig. 4. Metamodel of the LanGems Component Specification Language (LCSL)

Composition Language The composition of several language components and their adaptation for interoperation is externalised to a dedicated composition program. This ensures the independence of solitary language components and allows their flexible combination and adaptation during composition.

Fig. 5 depicts the metamodel of the composition language used in LanGems. It is connected with the metamodel for the LanGems Component Specification Language to describe the combination of several language components using the concepts defined in their specification. Every composition program defines a `Composer` which consists of a number of `Compositions`. Each `Composition` defines the mapping and adaptation of the provided interface of the `extended` component to the required interface of `generic` component. It comprises several `RoleBindings` which impose a played-by relation between a `Class` type of the extended component and a `Role` type of the generic component. The semantic integration of the role player is described by means of `RoleOperationBindings` for every `RoleOperation` defined in the `Role`. These bindings can be specified in the same `Semantics` formalism used for component semantics.

Composition Technique The third important part of the LanGems composition system is the composition tool and the composition technique it applies. The composition tool interprets the composition program and derives an integrated language with composed syntax and semantics.

The implementation of language components in LanGems is based on Java which does not provide concepts to represent roles and role bindings. Thus, we used a generic role pattern to implement role bindings. Fig. 6a) shows the result of applying this pattern to the role binding specified in Fig. 2. The resulting structure consists of three layers:

The Component's Type Interface implements the component's abstract syntax. For the role and the role player interfaces (`Action, LogAction`) are generated. These preserve the structural features (references, attributes, operations) of the types specified in the component model. Role operations are mapped to normal operations. This ensures the compatibility of the integrated language with the concrete syntax and seman-

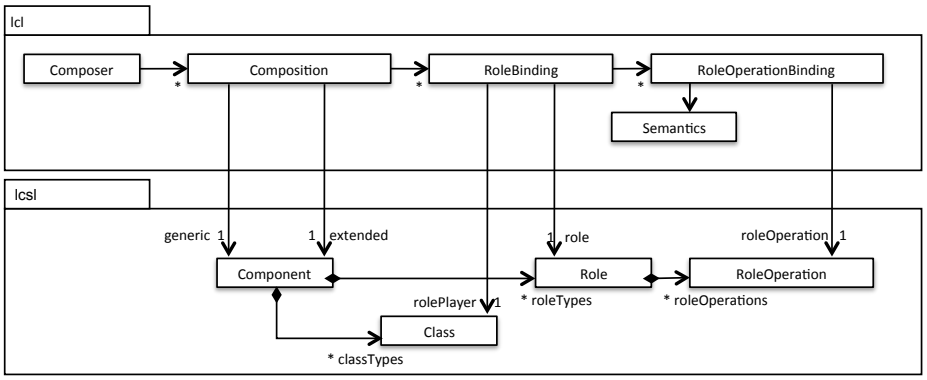


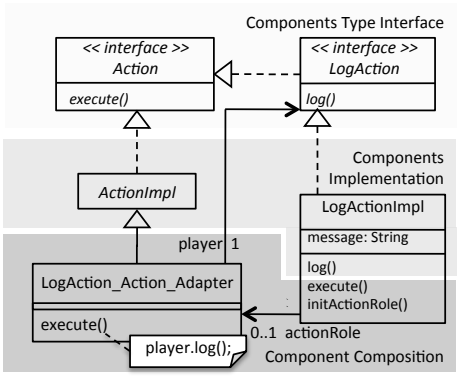
Fig. 5. Metamodel of the LanGems Composition Language (LCL)

tics defined within components. In accordance to the subtype-semantics of the played-by relation an implements-relationship between the interface of the role player to the role interface is introduced, to express their substitutability. This forces the role player `LogAction` to conform to the role's semantic constraints and implement the operation `execute()`. Since the implements-relationship is only introduced during language composition this does not interfere with our goal to keep the components decoupled.

The Component's Implementation contains the implementation of the component's type interfaces in the classes `ActionImpl` and `LogActionImpl`. These realise the functionality only relevant within the context of the according language component. This includes the EMF persistence functions for abstract syntax instances, the EMF-API for programmatic abstract syntax manipulation, and component semantics that are implemented operationally in Java.

Note that `ActionImpl` is abstract. This ensures, that role types can only be instantiated when bound to a concrete instance of their role-playing class types. The separation of the type interface and its implementation guarantees information hiding between components.

The Composition Implementation realises the integration of role and role player as defined in the composition program. It encapsulates the semantics adaptation specified in the composition program. The role binding in our example affects the implementation of the class `LogActionImpl` which also needs to implement the role-specific part of the interface `Action`. This is done by delegating all calls to operations defined for the role to the generated class `LogAction_Action_Adapter`. This class extends the abstract class `ActionImpl` and therefore preserves the properties and semantics defined within the role's component implementation layer. In addition the `LogAction_Action_Adapter` implements the missing role operations in accordance to the semantics adaptations defined in the composition program. To establish the context needed for semantics adaptation the adapter also needs to reference the role playing `LogAction` via the reference `player`.



a) Layered architecture of role pattern to implement component composition that preserves syntax and semantics of individual components

```
CONCRETE_SYNTAX eca_composed {

  ECAFile:= ”ECA” rules*;

  Rule := ”when” event
          ”if” condition
          ”do” action;

  Action := LogAction | ...

  LogAction := ”log” message;

...}
```

b) The composed syntax resulting from language composition

Fig. 6. Result of language composition for the ECA example

We needed to use a special variant of the adapter pattern for three reasons: First, classes need to be bound to multiple roles and preserve their predefined semantics. Therefore, the adapter is separated from the role-player implementation (`LogAction-Impl`) and derived from the role implementation (`ActionImpl`). Second, roles need to be bound to multiple role-players that realise their semantics contract. Therefore, the adapter is separated from the role implementation (`ActionImpl`) and references to the role-player (`player`). Third, the adapter needs to be instantiated when the role-player is instantiated (i.e., an expression of the type `LogAction` is recognised by the parser). Therefore, the Composition Implementation adds supplements to the role-player to control the creation of (`initActionRole()`) and the delegate to the adapter (`action-Role`).

The presented composition technique and the pattern used for its implementation allows for binding multiple role types to the same class type and also to have a role type bound to many class types. This provides the needed flexibility for integrating various language components. Furthermore, the composition is free of unanticipated effects on the structure of the involved components. This preserves the components' semantics and also realises semantics integration. For the given example the integration of composition semantics is straightforward, since the operational semantics defined in Java easily integrates with the implementation technology. Indeed for other semantics formalisms additional integration infrastructure is required. We have also investigated using OCL expressions to define semantics adaptations and component semantics. This implied generating additional code in the Component's Implementation Layer and the Composition Implementation that invokes an OCL interpreter which evaluates the OCL-based semantics. Since the code that needs to be generated followed the same pattern for all

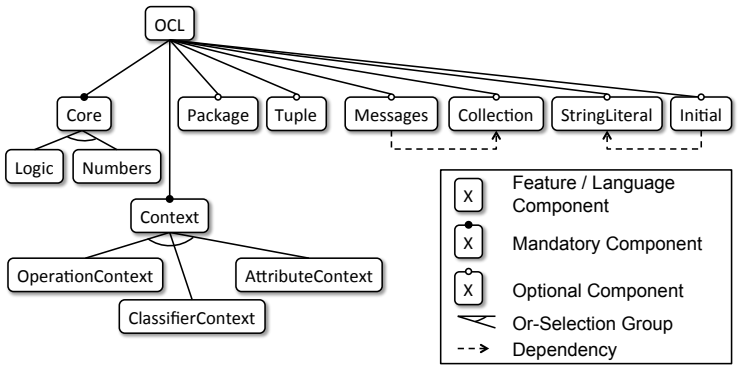


Fig. 7. OCL language components defined. Because there are very few direct dependencies between components, we have opted to present the component overview as a feature model [36]. Each feature stands for a specific language component, mandatory features represent components that must always be present to form a correct sub-language of OCL; optional features represent components that can be chosen or omitted. Additional dependencies between optional components are represented using dashed arrows

interpreter invocations, we could automate its generation during language composition. Currently, we are generalising this approach to support other semantics formalisms.

Figure 6b) shows the effect of the composition at the level of the concrete syntax. The composed grammar combines the syntax rules of both language components. Furthermore, an additional choice rule is generated in accordance to the role bindings given for the role type `Action`. This choice rule allows to choose between the concrete syntaxes defined for all role players (e.g. `LogAction`, ...).

4 Case Study: Modular OCL

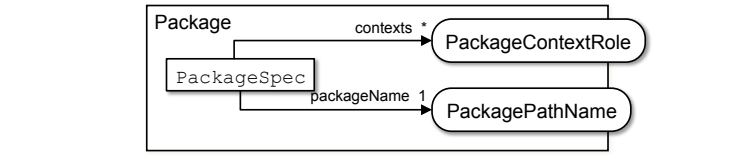
We have used our language modularisation technique to reimplement the Object Constraint Language (OCL) [37] in a modular fashion, based on our previous work presented in [38].[4] OCL has grown to become a quite large and complex language (the current version of the standard [37] comprises 232 pages). Still, a large number of extensions to the language have been and still are proposed, which are not covered by the current standard. In this context, modularising the language can provide benefits as it can enable constructing customised OCL variants based on a selection of core language features and extensions. Such languages will normally be smaller than the full OCL, reducing the learning effort involved in using them. Furthermore, a systematic modularisation of OCL makes it possible to integrate new extensions in a systematic manner, without having to rewrite the standard for each new extension. Thus, it is clear that modularising OCL is a sensible and useful undertaking. As OCL is a large and real-world language, its modularisation also provides realistic evaluation of the LanGems approach. Our modularisation produced a component structure (see Fig. 7). They all

[4] All details of this modularisation implementation can be found in [39].

contribute a specific part of OCLs abstract syntax, concrete syntax and semantics. The identified components are very similar to the OCL modules presented in [38]. However, using the novel language-modularisation technique presented in this paper brought a number of benefits for the modularisation of OCL:

1. *Explicit component interfaces.* The LanGems Component Specification Language provides two different types of nodes in the abstract syntax: class types and role types. Role types form the interface of a language component, explicitly representing the concepts the component requires to be defined in other components. In contrast, in the grammar-inheritance–based approach from [38], a language module's dependencies are well hidden within its specification. The explicit definition of component interfaces enabled by LanGems makes language modules much easier to understand and self-contained. Where in [38] to understand a language module one would have to read all the modules it depended on and understand how they mesh with the current module, here each module can be understood in isolation as its interface hides all details of other components.
2. *Improved decoupling of components.* Because language components come with an explicit interface, but also because the composition specification is separated from the definition of the components, the components are much more loosely coupled with each other. Figure 8 shows an example from the central component of the standard OCL definition—`Package`. This component essentially defines the structure of a complete OCL specification. Figure 8a) shows the definition based on LanGems. The `PackageContextRole` role type is a place holder for the different types of contexts to be supported. Which contexts will be available in a specific combination of components is decided in the composition model (see Fig. 9). In contrast, Fig. 8b) shows the same component as defined in [38]. Here, the different contexts to be available have to be explicitly named within the `std` component (see Lines 1 and 11). For each OCL derivative with a different set of contexts supported, this component needs to be rewritten. Notice, that in Fig. 8a) specific contexts are not mentioned at all. Their definitions will be inserted only after the composition program has been evaluated.
3. *Integration of semantics.* In contrast to the work in [38], the LanGems based modularisation also modularises the static semantics of OCL. This was possible because LanGems language components and role compositions allow the definition of arbitrary operational semantics for any abstract syntax graph. They work very similarly to attribute grammars [32], but with an encapsulation scope that distinguishes between component semantics and composition semantics. We intend to extend semantics modularisation to dynamic semantics of OCL using the term-rewriting system TOM [40]. This is also a case study to generalise the integration of semantics approaches with LanGems.

The strong decoupling and encapsulation also comes with its very own problems, however. In particular, encapsulation makes it more difficult to express operator priorities between operators defined in different components. As the OCL modularisation is based on the notion that 'everything is a navigation'—defining almost every OCL construct as a variation of the core navigation construct—priorities between the differ-



a)

```
01      CONCRETE_SYNTAX Package {
02        PackageSpec = ”package” packageName ”;”
03                        contexts+
04                      ”endpackage” ;
05      }
```

b)

```
01      grammar std extends invariant, operation {
02
03        oclFile = packageSpec;
04
05        packageSpec = ”package” packageName ”;”
06                        context+
07                      ”endpackage” ;
08
09        packageName = NAME  | pathName;
10
11        context = invariantContext | operationContext ;
12      }
```

Fig. 8. Package component constructed using a) LanGems and b) the approach from [38]

ent navigation operators were an important issue. In [38] priorities could be easily defined and redefined simply by overwriting productions inherited from other grammars. LanGems requires careful chaining of the individual components in the composition program to define an appropriate operator precedence. Even so, operator priorities can only be defined in blocks based on the components in which each operator has been defined. For example, consider component `A` defining operators `a` and `b` (with `a` having precedence over `b`) and component `B` defining operator `c`. Through appropriate chaining of the components we can easily express that `c` takes precedence over `a` and `b` or, alternatively, that both `a` and `b` take precedence over `c`. However, unless component `A` has made special provision for this case, we cannot express that `a` precedes `c` which precedes `b`. Notice though, that while grammar inheritance as used in [38] can express such constellations, it comes with its own problems. In particular, it is so open that a composition could even express (possibly accidentally) that `b` precedes `a`, which would probably cause problems with the remaining definition of component `A`

Because our current implementation of LanGems is based on the context-free parser used by EMFText [26], language component authors need to be handle conflicts with token definitions used in other components. This is so because tokens are resolved without reference to their grammatical context. Therefore, a token definition from component `A` could potentially overlap a token definition from component `B`, leading to failed matches for grammatical productions from `B`. This is a common issue with grammar composition, but could be solved, for example, by switching to a scannerless parsing technology (e.g., [28]). For similar technical reasons, the abstract syntax used in a language component currently needs to be very close to the concrete syntax of the language. Therefore, to produce a useful abstract syntax of OCL, we had to use an ad-

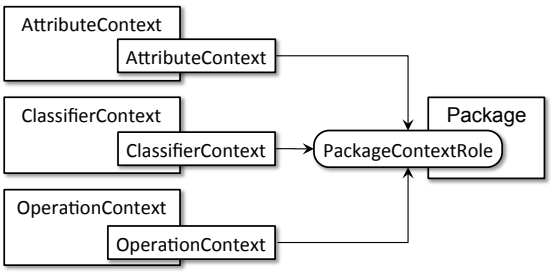


Fig. 9. LanGems composition program for standard OCL

ditional model-transformation step, as also discussed in [38]. Other approaches (e.g., [41]) show how to improve the level of abstraction of the abstract syntax without the need for manual coding of an additional model transformation. LanGems could be integrated with such approaches, and would gain from their benefits.

5 Related Work

Language modularisation has intensely been studied based on a modularisation of the concrete syntax of a language. Parser generators have originally been built to generate monolithic parsers from monolithic parser specifications [11]. However, as computer languages became more and more complex, and in particular with the advent of domain-specific languages, parser builders realised the benefits of modularity in language design. Thus, in recent years research on 'modular' or 'extensible' grammars has intensified and a number of different approaches have been developed that allow for a modular definition of parsers: Some approaches use so-called 'grammar inheritance', where one grammar can be extended by adding, refining, replacing, and removing individual productions [6, 7]. These approaches, however, only provide very initial support for modular language engineering, as they cannot guarantee that a parser can be generated directly from such a modular specification. This is a general issue with the LL(k) and LR(k) language classes typically considered for generating parsers; this issue is also shared by our approach. The problems involved have been mitigated somewhat with the introduction of special parsing approaches, such as context-aware scanning [42] or lexer and parser states [43] and the closely related delegating compiler objects [44]. Still, even with these approaches some languages cannot be successfully composed. This can only be guaranteed with parser generators supporting different classes of languages. Scannerless Generalised LR Parsing, as, for example, implemented in the Syntax Definition Formalism (SDF) [5], can parse the complete class of context-free languages. On the downside, it is a non-deterministic parser, so parsing time is no longer linearly bounded in the length of the text to parse. Parsing Expression Grammars (PEGs) [45, 46] are an example of grammars for a different language class that is also closed under composition, but can be parsed in linear time.

In contrast to modular parser specifications, our approach makes abstract syntax the central artefact of language modularisation. Monticore [41] is an approach that also

allows modularising the abstract syntax, but the description of the concrete syntax is the primary artefact. In contrast, in our approach, the abstract syntax is the primary artefact and is used as the basis for modularisation. Attribute grammars [32] are another modular approach for implementing the mapping from concrete syntax to abstract syntax allowing for modular abstract syntax based on a modular concrete syntax. Implementations that can also be coupled with modular parsers exist—for example, the JastAdd tool described in [9]. In [47] the authors present a generic approach to consolidate the capabilities of attribute grammars and their various extensions by functional meta-programming. Such meta-programming also supports semantics modularisation [48]. We plan a closer investigation of these techniques and their relation to role-based language composition for future work.

An interesting alternative approach to language modularisation has been presented by Cazzola and Speziale [49] who propose 'sectional' domain-specific languages. Such languages are constructed from language slices and roles. A language slice describes a certain feature of the language, while a language role provides information required for a specific step of language evaluation (e.g., parsing, type checking, code generation). In their paper, they further envision full language infrastructures to be generated from specifications composed from a number of slices and roles. Unfortunately, the work appears to still be somewhat immature and in progress, however it bears interesting parallels to our approach. We have chosen to focus on slices only, using specific fixed technology for the different possible roles and a fixed set of roles and requiring each slice (i.e., language component in our terminology) to be completely specified for each role. While this can be viewed as a limitation of our approach, it also has the benefit of simplifying implementation substantially.

The parsing techniques mentioned above are quite adequate for textual languages. Current graphical languages, however, are often "parsed"—that is, translated into an instance of some metamodel—by custom tools specifically built to go with the specific language. These tools are very often manually constructed, although some techniques, such as the Generic Modelling Environment (GME) [16] or the Generic Eclipse Modelling System (GEMS) [50], for generation of tools from an abstract language specification exist. To the best of our knowledge, however, no tools exist that would support generation of tooling for graphical languages from modular language specifications. Tools which follow an interpretive approach (e.g., MetaCase+ [18]), use a projective syntax editing technique (e.g., MPS [51]), or rely on generic graphical editors provide more flexibility w.r.t. language modularisation, but come with their own special limitations. A comprehensive classification and evaluation of these alternatives will require further research.

In Section 4 we mentioned that language modularisation helps the extension of OCL. Such customisation of language variants from extensible language specifications has also been discussed for instance for UML [52], for architecture description languages [53], for model-management languages [54], or for ontology languages [55]. Future research will have to provide methodologies for language family engineering where language customisation and language composition is driven by the requirements of language users. The concept of domain-specific metamodelling languages [56] can be an interesting starting point for such further research.

6 Conclusion

In this paper, we have presented LanGems—a role-based approach to component-based language development. In contrast to existing approaches, LanGems allows encapsulation of concrete and abstract syntax as well as semantics in one language component. The components are components in the sense of Szyperski's definition; that is, they are "[...] unit[s] of composition with contractually specified interfaces and explicit context dependencies only" [3]. The interfaces are defined using role types instead of standard metaclasses. These role types also define all of the context dependencies of a language module. Therefore, a LanGems language module "[...] can be deployed independently and [be] subject composition by third parties[...]" [3].

In applying this approach to the modularisation of OCL, we have found first of all, that it was possible to componentise this large, real-world language using our approach. However, the case study also showed some new problems that need to be attacked in future work. In particular, we need to find new ways for expressing operator precedence across language components balancing flexibility and component encapsulation.

Acknowledgments

This work was funded by the EC in the FP6 projects MODELPLEX and AMPLE, the FP7 project MOST, and by the German Ministry of Education and Research in the feasiPLe project.

References

1. van Deursen, A., Klint, P., Visser, J.: Domain-Specific Languages: An Annotated Bibliography. SIGPLAN Notices **35**(6) (2000) 26–36
2. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling. John Wiley & Sons Inc., Hoboken, New Yersey (2008)
3. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Second edn. Component Software Series. Addison-Wesley Publishing Company (2002)
4. Aßmann, U.: Invasive Software Composition. Springer-Verlag Inc., New-York (2003)
5. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The Syntax Definition Formalism SDF – Reference Manual. SIGPLAN Notices **24**(11) (1989) 43–75
6. Mernik, M., Lenič, M., Avdičaušević, E., Žumer, V.: The Template and Multiple Inheritance Approach Into Attribute Grammars. In: International Conference on Computer Languages. (1998) 102–110
7. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An Extensible Compiler Framework for Java. In: 12th International Conference on Compiler Construction. (2003) 138–152
8. Bravenboer, M., Visser, E.: Parse Table Composition Separate Compilation and Binary Extensibility of Grammars. First International Conference in Software Language Engineering: SLE 2008, Toulouse, France (2008)
9. Hedin, G., Magnusson, E.: The JastAdd System – An Aspect-Oriented Compiler Construction System. Science of Computer Programming **47** (2003) 37–58
10. Van Wyk, E., Bodin, D., Krishnan, L., Gao, J.: Silver: an Extensible Attribute Grammar System. 7th Workshop on Language Descriptions, Tools, and Analysis (LDTA'07) (2007)

11. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley (2006)
12. Kleppe, A.: Software Language Engineering. Pearson Education (2009)
13. Greenfield, J., Short, K.: Software Factories: Assembling Applications with Patterns, Frameworks, Models & Tools (2004)
14. Jouault, F., Bezivin, J.: KM3: a DSL for Metamodel Specification. 8th Conference on Formal Methods for Open Object-Based Distributed Systems, Bologna, Italy **4037** (2006) 171–185
15. OMG: MOF 2.0 core specification. OMG Document (January 2006) URL http://www.omg.org/spec/MOF/2.0.
16. Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., Volgyesi, P.: The Generic Modeling Environment. Workshop on Intelligent Signal Processing, Budapest, Hungary **17** (2001)
17. Amelunxen, C., Königs, A., Rötschke, T., Schürr, A.: MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. 2nd European Conference on Model Driven Architecture Foundations and Applications ECMDA-FA, Bilbao, Spain **4066** (2006) 361
18. MetaCase: MetaEdit+ - Workbench Users Guide. Online User's Guide (2009) http://www.metacase.com/support/45/manuals/mwb/Mw.html.
19. Bracha, G., Lindstrom, G.: Modularity Meets Inheritance. In: International Conference on Computer Languages, IEEE Computer Society (1992) 282–290
20. Taivalsaari, A.: On the Notion of Inheritance. ACM Computing Surveys **28**(3) (1996) 438–479
21. Reenskaug, T.: Working with objects. The OOram Software Engineering Method. Manning/Prentice Hall (1996)
22. Steimann, F.: On the Representation of Roles in Object-Oriented and Conceptual Modelling. Data Knowledge Engineering **35**(1) (2000) 83–106
23. Andersen, E.P.: Conceptual Modeling of Objects: A Role Modeling Approach. Ph.D. Thesis. Oslo, Norway, University of Oslo (1997)
24. Paton, N.W., ed.: Active Rules in Database Systems. Springer, New York (1999)
25. Budinsky, F., Brodsky, S.A., Merks, E.: Eclipse Modeling Framework. Pearson Education (2003)
26. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and Refinement of Textual Syntax for Models. In: Fifth European Conference on Model-Driven Architecture Foundations and Applications, ECMDA-FA. (2009) 114–129
27. Salomon, D., Cormack, G.: Scannerless NSLR (1) parsing of programming languages. Proceedings of the SIGPLAN'87 symposium on Interpreters and interpretive techniques **24**(7) (1989) 170–178
28. Visser, E.: Scannerless Generalized-LR Parsing. Technical Report, Programming Research Group, University of Amsterdam (P9707) (July 1997)
29. van den Brand, M., Scheerder, J., Vinju, J., Visser, E.: Disambiguation Filters for Scannerless Generalized LR Parsers. 11th International Conference on Compiler Construction, CC 2002 (2002)
30. Clark, T., Sammut, P., Willans, J.: Applied Metamodelling a Foundation for Language Driven Development (2nd Edition). Ceteva (2008) URL http://www.ceteva.com/book.html.
31. Winskel, G.: Formal Semantics of Programming Languages. The MIT Press (1993)
32. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory **5**(1) (1971)
33. Ekman, T., Hedin, G.: The JastAdd extensible Java compiler. SIGPLAN Not. **42**(10) (2007)
34. Van Wyk, E., Krishnan, L., Bodin, D., Schwerdfeger, A.: Attribute Grammar-based Language Extensions for Java. 21st European Conference on Object-Oriented Programming ECOOP, Berlin, Germany **4609** (2007) 575

35. Kastens, U., Waite, W.: Modularity and Reusability in Attribute Grammars. Acta Informatica **31**(7) (1994) 601–627
36. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-0211990, Software Engineering Institute (1990)
37. Object Management Group: UML 2.0 OCL specification. Object Management Group Available Specification, formal/06-05-01 (May 2006) URL http://www.omg.org/spec/OCL/2.0/PDF/.
38. Akehurst, D., Zschaler, S., Howells, G.: OCL: Modularising the language. In: Ocl4All: Workshop at MoDELS 2007, Electronic Communications of the EASST. Volume 9. (2008)
39. Thieme, N.: Modulare Redefinition von OCL. Diplomarbeit, Technische Universität Dresden, Germany (2008) In German.
40. Moreau, P., Ringeissen, C., Vittek, M.: A Pattern Matching Compiler for Multiple Target Languages. 12th Conference on Compiler Construction, Warsaw, Poland (2003) 61–76
41. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: Modular Development of Textual Domain Specific Languages. In: 46th International Conference on Objects, Models, Components, Patterns (TOOLS-Europe). (2008)
42. Van Wyk, E., Schwerdfeger, A.C.: Context-aware scanning for parsing extensible languages. In: 6th International Conference on Generative Programming and Component Engineering (GPCE'07), New York, NY, USA, ACM (2007) 63–72
43. Clark, C.: Newlines and Lexer States. SIGPLAN Notices **35**(4) (2000) 18–24
44. Bosch, J.: Delegating Compiler Objects: Modularity and Reusability in Language Engineering. Nordic J. of Computing **4**(1) (1997) 66–92
45. Ford, B.: Parsing Expression Grammars: A Recognition-based Syntactic Foundation. In: 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04), New York, NY, USA, ACM (2004) 111–122
46. Grimm, R.: Better Extensibility through Modular Syntax. In: 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'06), New York, NY, USA, ACM (2006) 38–51
47. Lämmel, R., Riedewald, G.: Reconstruction of paradigm shifts. (1999) INRIA, ISBN 2-7261-1138-6.
48. Lämmel, R.: Declarative aspect-oriented programming. (1999) 131–146
49. Cazzola, W., Speziale, I.: Sectional Domain Specific Languages. In: Workshop on Domain-Specific Aspect Languages (DSAL'09), co-located with AOSD'09. (2009) 11–14
50. White, J., Schmidt, D.C., Mulligan, S.: The Generic Eclipse Modeling System. In: Model-Driven Development Tool Implementer's Forum, TOOLS '07. (2007)
51. Jetbrains: MPS - Meta Programming System. MPS Website (2009) http://www.jetbrains.com/mps/index.html.
52. Atkinson, C., Kuhne, T.: The Essence of Multilevel Metamodeling. Proccedings of the 4th International Conference on the Unified Modeling Language (2001) 19–33
53. Völter, M.: A Family of Languages for Architecture Description. In Proceedings of the 8th Workshop on Domain-Specific Modeling (2008)
54. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.: The Epsilon Generation Language. In Proceedings of the 4th European Conference on Model Driven Architecture - Foundations and Applications (2008) 1–16
55. Wende, C., Heidenreich, F.: A Model-based Product-Line for Scalable Ontology Languages. In Proceedings of the 1st International Workshop on Model-Driven Product Line Engineering (2009)
56. Zschaler, S., Kolovos, D., Drivalos, N., Paige, R., Rashid, A.: Domain-Specific Metamodelling Languages for Software Language Engineering. In: 2nd Int'l Conf. on Software Language Engineering (SLE). (2009)