LCL
A Graphical Meta-Language for Specification of Language Constraints

Terje Gjøsæter
Department of ICT, University of Agder, Grimstad, Norway

Keywords: Meta-languages, Meta-modelling, Language Constraints.

Abstract: The Object Constraint Language (OCL) is commonly used for constraints in meta-model-based language specifications. However, it may be advantageous to have a domain-specific constraint meta-language optimised for language specifications. A survey of OCL usage in language specifications has been performed, in order to gain an understanding of common constraint patterns. This is used as a starting point for defining a new meta-language for language constraints, Language Constraint Language (LCL), that has an intuitive graphical syntax.

1 INTRODUCTION

There has recently been much focus on *domain-specific languages* (DSLs) - computer languages focusing on a particular problem domain, that can be rapidly created and deployed (Kelly and Tolvanen, 2008). This puts strong demands on the efficiency and usability of tools and technologies for creating computer languages. A popular approach is to generate language tools from language *specifications*. Computer language specifications are (more or less) formal and complete descriptions of a computer language. They can be used as blueprints for manually developing tools for handling the language, or if a specification is sufficiently formal and complete, it may be used for automatically generating language tools. These tools may for example be code or diagram editors, and interpreters, compilers or code generators for executing the statements in the language instance or transforming them into executable form.

There are different approaches to creating tools for computer languages. *Compiler theory* concerns developing *compilers*, i.e. tools for turning a language instance into another (usually executable) form. It has its strength in defining optimised compilers for large textual general purpose languages. On the other hand, the focus among language designers is shifting towards creating small DSLs. These languages may have a graphical or textual presentation (concrete syntax), and they are often based on existing languages and may be preprocessed / embedded / transformed into other languages for execution, instead of being compiled with a traditional compiler. This new trend requires new approaches, therefore speed of tool development and agile approaches to language development are increasingly important. *Model-driven architecture* (MDA) is a general modelling approach, that also has some advantages when it comes to defining DSLs. An important advantage of MDA for language development is that it provides the language designer with support for rapid development and automatic prototyping of language support tools, and allows for working on a high level of abstraction. This approach allows the language designer to focus on the language being developed, while still being able to use the definition for generating tools such as editors, validators and code generators.

Following the MDA approach to language development, it is important that the technologies used are well suited to the language development domain, and that they allow the language designer to operate on a high level of abstraction. In the following sections, we will examine technologies that are being used for defining constraints in language specifications, and in particular the most commonly used of these that is the Object Constraint Language (OCL) (OMG, 2005). Based on our findings on the usage of constraints in language specifications, we will propose an alternative approach to defining constraints that is customised for language design.

The rest of the article is organised as follows: Section 2 covers general background of language specification, and Section 3 covers constraints in more detail. In section 4, we describe a survey on usage of

Gjøsæter T..
LCL - A Graphical Meta-Language for Specification of Language Constraints.
DOI: 10.5220/0005250603290337
In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2015), pages 329-337
ISBN: 978-989-758-083-3

constraints in language specifications, that leads to two different candidate approaches for defining a new language constraints language. In section 5, we propose an alternative approach for handling language constraints; and in sections 6, the benefits and limitations of the proposed language are discussed. Related work is discussed in 7, and finally 8 contains a summary of the article as well as plans for future work.

2 LANGUAGE SPECIFICATION

There are different ways to divide the aspects of a language specifications. In traditional compiler theory, we often refer to *concrete* and *abstract syntax*, and *static* and *dynamic semantics*. In meta-modelling, the structural aspect is often referred to as the *meta-model*, and we may find cases where the term meta-model also include other language aspects like constraints. In (Nytun et al., 2006), a language definition is said to consist of the following aspects: *Structure*, *Presentation* and *Behaviour*, and we also include *Constraints* and *Mapping* for completing the picture.

Structure defines the constructs of a language and how they are related. This language aspect is often named *abstract syntax*. *Constraints* bring additional constraints on the structure of the language, beyond what is feasible to express in the structure itself. This is related to what is called *static semantics* in traditional compiler theory. *Presentation* defines how instances of the language are presented to the developer. This can be the definition of a graphical or textual *concrete syntax*. *Behaviour* explains the *dynamic semantics* of the language. This can be a transformation into another language (denotational or translational semantics), or it defines the execution of language instances (operational semantics). *Mapping* binds together specifications for structure with specifications for presentation and for behaviour.

In a meta-model-based language specification environment, it is common that each language aspect is handled by a separate domain-specific language, also called a *meta-language*, with tools for generating the needed code. In the popular environment Eclipse, structure may for example be defined by the ECORE language, and code generation handled by the Eclipse Modeling Framework (EMF)(Steinberg et al., 2008). There are several plug-ins available that cover other language aspects. These typically provide their own aspect-specific meta-language(-s) and tools that integrate the code generated from the aspect specification in the EMF-based structure code, thus providing a complete editor/execution environment for the language being developed.

3 LANGUAGE CONSTRAINTS

Language constraints can put limitations on the structure of a well-formed instance of the language. This aspect of a language definition mostly concerns logical rules or constraints on the structure that are difficult to express directly in the structure itself. Neither meta-models nor grammars provide all the expressiveness that is needed to define the set of wanted language instances. However, there is an overlap between the structure and constraint aspects of a language. Some language features may obviously belong to one of them, but many features could belong to either of them, depending on choice or on the expressiveness of the technology used to define the structure.

What we typically want to express with constraints in a compiler-based language specification is logical rules and restrictions (static semantic conditions) related to elements of the language structure. Often, these constraints are expressed in code using a general purpose programming language, and in some cases in a separate logic language. These logical rules may be attached to attributes in an attribute grammar.

IBM's SAFARI (an IDE development meta-tool) and MontiCore (a framework for development of domain-specific languages) have some support for constraint handling. Many tools handling attribute grammars can also handle constraints defined on the attributes, e.g. with code or logical expressions.

While meta-models are constructive definitions of what objects a language instance can consist of, constraints allow to narrow down the possible instances of a meta-model class. A meta-model constraint expressed is thereby usually written in the context of a class, and only constrains the set of possible instances (objects) of this class. A constraint forms a logical expression. It takes an instance of the context class as input and evaluates to a boolean value, assessing the instance as either a valid, or an invalid instance. Only the models that exclusively consist of valid objects are valid language instances.

In general-purpose modelling as well as meta-modelling, model constraints allow the modeller to restrict the possible valid instances of a model by defining logical expressions over elements of the model. This is commonly done in e.g. UML with the Object Constraint Language (OCL) (OMG, 2005). OCL is a formal language for describing expressions on UML models. These expressions typically specify invariants and pre- and post- conditions that must hold for the system being modelled. OCL can also express queries over objects described in a model. OCL is designed to present the expressiveness of predicate logic, in a programming language like syntax. Re-

lated tools allow to check whole models or single objects, based on the constraints associated with the model's meta-model classes. Meta-model-based language tools usually do not check a model within a separate tool, but are integrated into model editors. Plugins that provide support for OCL constraints in Eclipse include MDT OCL (Willink, 2012), Dresden OCL (Birgit Demuth and Claas Wilke, 2009), and the EMF Validation Framework (EMF-VF).

There are some important issues of constraints that are not handled by the OCL language, i.e. the seriousness of breaking a constraint, and when it should be checked. These issues will have to be dealt with in the implementation.

3.1 Constraints in the UML 2 Infrastructure Specification

The UML 2.3 Specification consists of two complementary specifications: Infrastructure and Superstructure. The UML infrastructure specification defines the foundational language constructs required for UML 2.3, and the UML Superstructure defines the user level constructs required for UML 2.3 (OMG, 2007). The primary language used for defining constraints in this specification is OCL. The following quote from (OMG, 2007) describes the use of constraints in the UML infrastructure specification:

> *The well-formedness rules of the metaclass, except for multiplicity and ordering constraints that are defined in the diagram at the beginning of the package sub clause, are defined as a (possibly empty) set of invariants for the metaclass, which must be satisfied by all instances of that metaclass for the model to be meaningful. The rules thus specify constraints over attributes and associations defined in the metamodel. Most invariants are defined by OCL expressions together with an informal explanation of the expression, but in some cases invariants are expressed by other means (in exceptional cases with natural language).*

The usage of OCL constraints in the UML specification is also supported by auxiliary functions defined in the specification.

3.2 Constraints in the SDL Specification

The SDL specification (ITU-T, 1999) is organised by topics described by an optional introduction followed by titled enumeration items for:
a) Abstract grammar, described by abstract syntax and static conditions for well-formedness.
b) Concrete textual grammar. This grammar is described by the textual syntax, static conditions and well-formedness rules for the textual syntax, and the relationship of the textual syntax with the abstract syntax.
c) Concrete graphical grammar, described by the graphical syntax, static conditions and well-formedness rules for the graphical syntax, the relationship of this syntax with the abstract syntax.
d) Semantics, that gives meaning to a construct, provides the properties it has, the way in which it is interpreted and any dynamic conditions which have to be fulfilled.
e) Model, that gives the mapping for notations that do not have a direct abstract syntax and modelled in terms of other concrete syntax constructs.
f) Examples.

The SDL specification does not use OCL or other dedicated constraints languages for constraints. The constraints are instead expressed in English and take the form of sentences referring to elements of the grammars. They are mostly well-formedness rules for the concrete grammars, but there are occasionally also constraints for the abstract syntax. However, it must be noted that the document "ITU-T Recommendation Z.100 Annex F: SDL Formal Semantics Definition" (ITU-T, 2007) provides formal semantics for SDL including a formal description of constraints.

4 CATEGORISATION OF CONSTRAINTS USAGE

A survey of the usage of constraints in two different language specifications, the UML infrastructure specification (OMG, 2007) and the SDL specification (ITU-T, 1999), has given insights that may be taken as a starting point for creating a higher abstraction level approach to language constraints. Two different approaches are tested, the first approach is focused on classifying and grouping the constraints with the purpose of finding common semantic patterns that can be applied to a language specification. The second approach investigates possible structural commonalities among constraints that can be exploited.

4.1 Semantic Approach

The first approach attempted was based on grouping and classifying constraints into sets of common constraint types that could be used as a basis for a library of common constraint patterns that could be ap-

plied to a language specification. If successful, this approach could significantly lower the complexity of defining language constraints by allowing the language developer to pick and apply the relevant constraint patterns for the language to be developed. We started with an initial set of expected constraints categories, based on previous experience with language constraints: *Name-space-related constraints*, *Type-related constraints*, *Structural constraints* and *Syntactic matching*.

By going through the constraints in the UML Infrastructure specification (OMG, 2007) and attempting to sort them into these categories, more categories were discovered that were added to the initial set. As the survey progressed, the categories were re-evaluated and adjusted until a stable set of categories was established. Then, the SDL specification was examined and used for verifying the adequacy of the set of categories.

The total number of constraints in the UML infrastructure specification is 67. The main categories of constrains discovered in the specification are:

Namespace handling (5)
Example: *All the members of a Namespace are distinguishable within it (9.14.2.1 of (OMG, 2007)).*

Type handling (9)
Example: *If this operation has a return parameter, type equals the value of type for that parameter. Otherwise type is not defined (11.8.2.6 of (OMG, 2007)).*

Structural constraints (25)
Example: *Generalization hierarchies must be directed and acyclical. A classifier cannot be both a transitively general and transitively specific classifier of the same classifier (9.19.1.1 of (OMG, 2007)).*

Context constraints (1)
Example: *Subsetting may only occur when the context of the subsetting property conforms to the context of the subsetted property (11.3.5.3 of (OMG, 2007)).*

Helper functions for constraints (not counted)
Example: *The query allFeatures() gives all of the features in the namespace of the classifier. In general, through mechanisms such as inheritance, this will be a larger set than feature (9.4.1.1 of (OMG, 2007)).*

Dynamic semantics constraints (27)
Example: *If this operation has a return parameter, isOrdered equals the value of isOrdered for that parameter. Otherwise isOrdered is false(11.8.2.3 of (OMG, 2007)).*

We note that three new categories are identified (the last three in the list above), and one expected category that is removed (syntactic matching) after a lack of related constraints identified.

The SDL specification (ITU-T, 1999) has also been examined, and when it comes to abstract syntax, the constraints fall into the categories above. However, it is noticeable that while the UML specification uses constraints to cover the limitations of MOF for defining structure, the SDL specification uses constraints to cover the limitations of grammars, for example for simulating associations with multiplicity. For concrete textual and graphical syntax, there are two additional categories of constraints detected:

Location of elements
Typically takes a form like *<element A> is to the right of <element B>* or *<element A> is inside <element B>*.

Mapping handling
Typically takes the form *<concrete element> represents <abstract element>*.

However, we encountered complications when trying to divide the categories further into simple (positive or negative) patterns that can be handled by a new constraint language. Most of the constraints are falling into a rather vague structural constraints category, where it is difficult to isolate common patterns. With this in mind, a different approach is attempted, as described in the following.

4.2 Structural Approach

The majority of the analysed constraints are in the form of implications, with object patterns on the right- and left-hand sides of the implication, in many cases combined with logical operations. By analysing the constraints, we find that more complex constraints involving logical operations like AND and OR, can be handled by rewriting or splitting up the constraints.

We note that constraints tend to be *local*, concerning an object and its attributes and references; but it may also be more complex, making it necessary to look at a bigger picture. For the non-local constraints, some require looking at the *parent* node in the graph structure, some require *iterative* navigation to parent nodes, and a few are completely *global*, requiring more complex querying of objects. Constraints may also contain *external* references, referring to something outside the specification, e.g. filenames, modules, or editors. However, this type of constraints can also not be handled by OCL. In the UML specification, the initial quick survey detected 86 local constraints, 29 parent constraints, 3 iterated constraints

and 20 global constraints. However, the constraints that were initially put into the global category tended to be the most complex and difficult to understand, and since global patterns are harder to express as object diagrams, we performed a second closer analysis of the initially thought global ones that uncovered that all of them actually more correctly fall into either local, parent or iterated categories, and in a few cases the statement is not a constraint at all but e.g. a explanatory statement for the reader. The final results are 68% local patterns, 22% parent patterns, 6% iterated patterns, 0% global patterns and 4% not a testable constraint. A brief survey of the SDL specification shows a similar trend in the distribution of the results, although there are less local patterns and more parent and iterated patterns found in this case: 50% local patterns, 30% parent patterns, 20% iterated patterns, 0% global patterns. Note that non-constraints statements were not counted in the SDL specification. This indicates that a structural approach may handle the great majority of real-world language constraints, and therefore be valid for real-world usage.

4.3 Do we Really Need a New DSL for Language Constraints?

While OCL is more specialised for the task of defining constraints than general-purpose programming languages, it may still be argued that we are on a relatively low abstraction level, using a quite general logical language for something much more specific. OCL has a wide feature set that handles a wide range of model types, and many of these features will normally not be needed in a language specification.

Although OCL clearly is up to the task of defining constraints in a language specification, it also makes sense to create a domain-specific language not because we need features that general purpose languages cannot offer, but because we would like to reduce unneeded complexity. Therefore, a DSL for language constraints called LCL (Language Constraint Language), is proposed. It aims to be a simple, intuitive and easy to use alternative to OCL (or code), that covers the needs of the language developer.

4.4 The Chosen Approach

We have found that simple series of object pattern implications will cover the majority of the cases encountered during this survey. A typical constraint will then consist of 3 parts; *context pattern*, one or more *condition patterns*, and finally a *check pattern*. The context pattern part is setting the context or applicability criteria for the constraints in much the same way as in OCL. An advantage over OCL may be that we are able to express more complex contexts than OCL's single element context. The context pattern is always positive, while the condition patterns may be positive or negative as needed, to express that the given pattern must or must not be present in the model for the implication to be valid. These patterns functions to restrict the possible sets of elements that the constraint applies to and may also introduce additional connected elements. The last pattern in the implication series is the check pattern. This functions as an evaluation point where we get the result of the constraint evaluation; *True* if the implication series is valid, and *False* otherwise. Based on the evaluation result, a response to the user is generated. This approach is further described in the following section.

5 A NEW META-LANGUAGE FOR LANGUAGE CONSTRAINTS

From examining OCL usage in language specifications, we gained insights that are used as a foundation to define a new DSL for language constraints. A constraints language based on object patterns is proposed, that also supports handling of constraints checking issues like when to check, and what to do if a constraint is broken.

5.1 Structure of LCL

Figure 1 shows the basic structure of the Language Constraints Language (LCL). The two main parts of the language are the `Constraint` with its connected `ObjectPattern` classes, and the `CheckHandling` class. The attributes of the `CheckHandling` class determine when a constraint should be checked, how severe a breakage is, what to do if it is broken, and an optional message to the user.

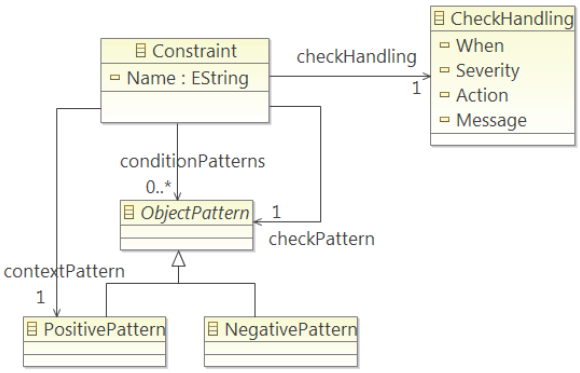

Figure 1: The Structure of the Language Constraints Language.

The `ObjectPattern` part is here abbreviated into a single class for reasons of space and clarity, but does actually represent a simple object description language similar to UML object diagrams, enhanced with recursive associations for iterated patterns.

The `Constraint` has three associations corresponding to the three elements of the constraint; the context, the conditions and the check. The `context pattern` may only reference a `PositivePattern`, that represents an object diagram pattern that has to be present in the model for the constraint to apply. The `conditionPatterns` association points to a sequence of positive or negative patterns that further details the scope of the constraint in the form of an implication chain. The `checkPattern` finally provides the positive or negative pattern that will be evaluated to be true or false based on the previous implication chain.

5.2 Presentation of LCL

A graphical presentation for LCL may take the form of a sequence of boxes containg object patterns, as shown in Figure 2. Each box contains an object diagram representing the relevant pattern, with a plus or minus sign in the corner of the box determining if it contains a positive or negative pattern.

The illustrated constraint is taken from section 9.21.1 of (OMG, 2007): "If a NamedElement is not owned by a Namespace, it does not have a visibility." In OCL, it is expressed as: `namespace->isEmpty() implies visibility->isEmpty()`. In the LCL example, name of the constraint is given in the top of the diagram, "NamedElementVisibilityConstraint". The box on the left hand side contains the context pattern. This pattern is positive, as context patterns always have to be. This is signified by the "+" in the upper right corner. The "+" means that the constraint applies if the pattern is found in the given model that it is attached to. In this example, the context is the existence of a `NamedElement`. The NamedElement is given the variable name `a` so we can refer to it in later patterns. The next pattern is a negative pattern, expressing that the constraint applies if the NamedElement from the previous pattern does not contain an association to a NameSpace. The last pattern is the check pattern, giving the final criteria that the constraint will be evaluated based on. It states that given the previous restrictions, the `Visibility` attribute of the `NamedElement` must have a value NULL. Note that the check handling parameters are not shown in the constraint examples provided here.

5.3 Semantics of LCL

In a nutshell, the semantics of LCL can be separated into two parts, the evaluation of the constraints, and the handling of checking and reporting results. There are 5 main steps in handling a constraint:

- CheckHandling: based on the checkHanding properties of the Constraint, evaluation of the Constraint is triggered.
- Context: search the model for objects that fit the criteria given by the contextPattern, and select these for further evaluation.
- Conditions: for each pattern in the conditionPatterns sequence, restrict the set of objects accordingly.
- Check: evaluate the final implication towards the checkPattern, and return a True or False result accordingly.
- Report: based on the checkHanding properties of the Constraint, a message to the user is generated, and appropriate action is triggered.

Interpretation of positive and negative patterns:

- A *positive* pattern is *valid* if and only if all elements of the pattern are present in a structurally equivalent form in the model to be evaluated.
- A *negative* pattern is *invalid* if and only if all elements of the pattern are present in a structurally equivalent form in the model to be evaluated.

On object names in patterns:

- Names are introduced to identify objects between patterns in a constraint.
- Names are limited in scope to a single constraint.
- Different object description $\widehat{=}$ different objects unless name is equal.
- Negative patterns cannot introduce object names.
- Need also object description reference! (i.e. refer to more info of an object)

5.4 Additional Examples of LCL Constraints

The first example, taken from section 11.8.2 of (OMG, 2007), shows how a complex constraint can be expressed as two different constraints in LCL: "If this operation has a return parameter, type equals the value of type for that parameter. Otherwise type is not defined." In OCL, it is expressed as: `type = if returnResult()->notEmpty() then returnResult()->any().type else Set endif`.

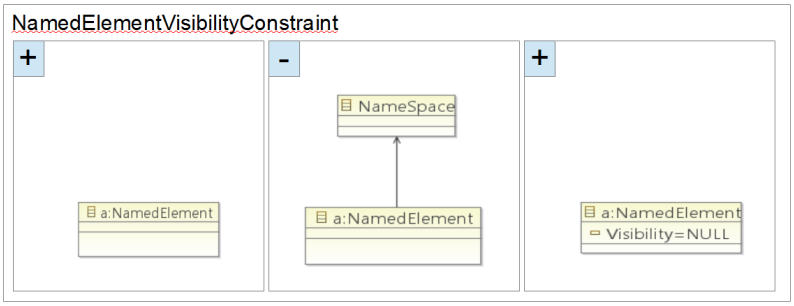

Figure 2: Example constraint in LCL.

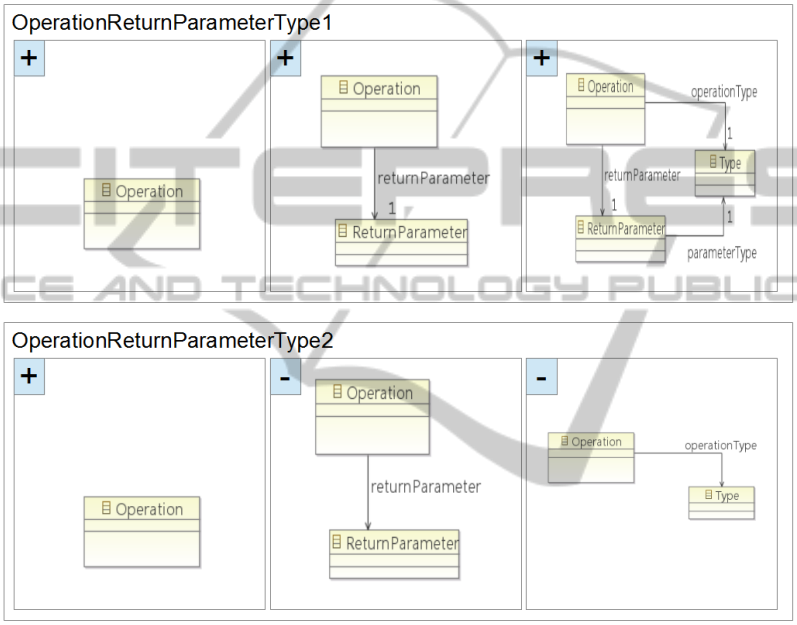

Figure 3: Type checking constraint in LCL.

In LCL, we divide it into two constraints as shown in Figure 3, both first setting the context to Operation objects. The first limits the set to operations having a returnParameter, while the second constraints covers operations without such a parameter. The last part of each constraint then in the first case requires the associated type to be the same, while in the second case, the operation should not have a type, as given in the corresponding negative pattern.

The second example is taken from section 13.1.8 of (OMG, 2007)and states that "A stereotype must be contained, directly or indirectly, in a profile." OCL solves this using a specialised function, expressed as `profile = self.containingProfile()`. In LCL (see Figure 4), we can use this example to demonstrate both a minimal constraint with only two patterns, the context and the check pattern; and use of the iterated association shown using an association with a star in the middle. This means that the element

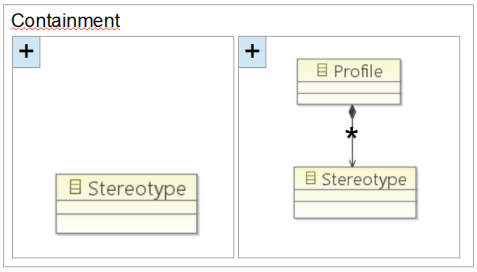

Figure 4: Containment constraint in LCL.

must be directly or indirectly associated with the other element. In this case, showing that a Stereotype must be directly or indirectly *contained* in a Profile.

The third example from section 9.19.1 of (OMG, 2007), states that "Generalization hierarchies must be directed and acyclical. A classifier cannot be both a transitively general and transitively specific classifier of the same classifier." This is expressed in OCL

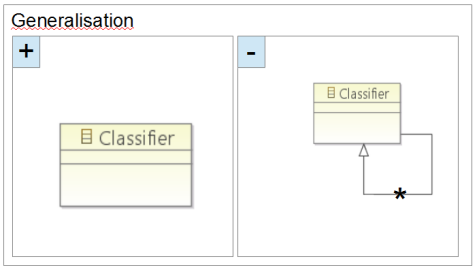

Figure 5: Generalisation hierarchies constraint in LCL.

as `not self.allParents()->includes(self)`. In LCL, it can be expressed as shown in Figure 5, using again an iterated connection, but this time an iterated generalisation in a negative pattern showing that a Classifier can not directly or indirectly generalise itself.

6 DISCUSSION

The chosen solution primarily focusses on conceptual issues and does not provide a complete language with all aspects fully specified. However, refinement of LCL into a complete language specification, and further implementation into a usable language, would not be very difficult given the relative simplicity of the provided solution.

The solution provides the possibility to handle language constraints on a higher level of abstraction, and allows the language developer to define when to check each constraint and what to do if it is broken. Solutions like adding higher abstraction level functions to OCL (i.e. LCL as an internal DSL in OCL) or to add support for constraint check handling could also partly solve the problem, but it is considered important that a constraints language should be as simple to use as possible, and that is a strength of defining LCL as an external DSL. If a wanted constraint is not possible to express with the implications and object diagrams supported by LCL, should then the use of OCL be allowed? It would be constructive to first determine if the newly requested constraint is actually one that could be split up into manageable parts, or if wanting the constraint is the result of bad design decisions by the language designer. We note for example that expressions support is a feature that is present in OCL but missing from LCL. However, while it is clearly needed in a general purpose constraints language, the survey of language constraints shows that the need for expressions in language constraints is small and the constraints can usually be expressed in other ways.

The approach is intended to make defining language constraints easier. With a graphical approach and a high level of abstraction, it should be less prone to bugs than by using OCL or similar logical languages, or even general purpose programming languages. A full logical language may in most cases be overkill for the issues to be handled by language constraints, when a more simple and easy to understand language would be sufficient. This design should help the language developer avoid errors and bugs related to malformed or logically incorrect constraints, and will lower the requirement for programming skills of the language developer and allow him to operate on a higher level of abstraction. The chosen approach would be particularly suited for DSL developers; a graphical approach is more easy to grasp for non-programmers and therefore facilitates communication with domain experts during development of DSLs. An additional advantage to LCL is that it is easy to learn as it is just a minor extension to an already well-known object diagram notation.

Real-world usage is needed to determine if this approach is able to cover the needs of a language developer in its current form, or if it has to be extended, e.g. with more complex implication patterns.

7 RELATED WORK

A limited but significant amount of related work can be found. Among others, there are articles related to language workbenches and their technologies that are worth noting as they may also cover constraints support in these workbenches. Research related to domain-specific constraint(-like) languages outside of the language domain may also be interesting for comparison across domains. Finally, research on general purpose constraint languages may also be relevant.

In the first category, *The State of the Art in Language Workbenches* by Erdweg, van der Storm, et al. gives an interesting overview of the participants in the 2013 Language Workbench Competition, where it describes the features supported in several language workbenches and gives some insights on how the different workbenches support constraints; ranging from no support, to support for specific types of constraints like type enforcement through dedicated declarative languages, but in many cases ether relying on programmatic handling or semantically rich meta-models (Erdweg, Sebastian and van der Storm, Tijs and Völter, Markus and Boersma, Meinte and Bosman, Remi and Cook, William R and Gerritsen, Albert and Hulshout, Angelo and Kelly, Steven and Loh, Alex and others, 2013). Visser and Eelco's *Separation of Concerns in Language Definition* propose an approach to

separation of concerns in language definition that includes high-level declarative meta-languages that includes constraints (Visser, 2014).

In the second category, Mens, van der Straeten and d'Hondt cover model consistency checking in *Detecting and resolving model inconsistencies using transformation dependency analysis* (Mens et al., 2006) with an interesting approach using transformation rules that have some similarity to the graphical presentation of LCL, although it has a different goal from the current issue of constraints enforcement in language specifications.

The third category includes Jaffar and Mahler's *Constraint logic programming: a survey* that gives an overview of constraint logic programming, from which domain-specific constraint languages may find inspiration (Jaffar and Maher, 1994).

8 SUMMARY AND FUTURE WORK

A survey of OCL usage in language specifications has been performed, in order to gain an understanding that is used to define a new approach for language constraints. A solution based on structured object pattern implications is proposed, that also supports handling of constraints checking issues like when to check, and what to do if a constraint is broken. The proposed solution allows language designers to focus on language features rather than complex logical expressions. It may be particularly useful for DSL developers because of its familiar graphical syntax that make constraints in LCL both easy to understand and easy to explain to stakeholders that may be less familiar with logical expression languages.

There are still open issues left for future work. A full formal specification of all aspects of the proposed constraints language is needed. A textual concrete syntax is being developed as an alternative to the graphical diagram-based presentation. A prototype implementation of LCL is already ongoing, adding constraints support to the LanguageLab language workbench (Gjøsæter and Prinz, 2012) that will be tested on Master students studying computer language theory and design next year. Based on experiences from this prototype, the approach will be further refined and adapted to the needs of DSL developers.

REFERENCES

Birgit Demuth and Claas Wilke (2009). Model and Object Verification by Using Dresden OCL. In *Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice, July 25-31, Ufa, Russia, 2009*, page 81. Ufa State Aviation Technical University, Ufa, Bashkortostan, Russia.

Erdweg, Sebastian and van der Storm, Tijs and Völter, Markus and Boersma, Meinte and Bosman, Remi and Cook, William R and Gerritsen, Albert and Hulshout, Angelo and Kelly, Steven and Loh, Alex and others (2013). The state of the art in language workbenches. In *Software Language Engineering*, pages 197–217. Springer.

Gjøsæter, T. and Prinz, A. (2012). Languagelab 1.1 user manual. Technical report, University of Agder.

ITU-T (1999). SDL - ITU-T Specification and Description Language (SDL-2000). ITU-T Recommendation Z.100.

ITU-T (2007). *Recommendation Z.100 Annex F: SDL Formal Semantics Definition*. International Telecommunications Union (ITU), Geneva.

Jaffar, J. and Maher, M. J. (1994). Constraint logic programming: a survey. *The Journal of Logic Programming*, 19–20, Supplement 1(0):503 – 581. Special Issue: Ten Years of Logic Programming.

Kelly, S. and Tolvanen, J.-P. (2008). *Domain-Specific Modeling*. Wiley-Interscience.

Mens, T., Van Der Straeten, R., and D'Hondt, M. (2006). Detecting and resolving model inconsistencies using transformation dependency analysis. In *Model driven engineering languages and systems*, pages 200–214. Springer.

Nytun, J. P., Prinz, A., and Tveit, M. S. (2006). Automatic generation of modelling tools. In Rensink, A. and Warmer, J., editors, *ECMDA-FA*, volume 4066 of *Lecture Notes in Computer Science*, pages 268–283. Springer.

OMG (2005). *OCL 2.0 Specification*. Object Management Group. ptc/2005-06-06.

OMG (2007). *UML Infrastructure Specification, V2.1.2*. Object Management Group. ptc/06-10-06.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2008). *EMF: Eclipse Modeling Framework*. Eclipse Series. Addison-Wesley Professional, second edition.

Visser, E. (2014). Separation of concerns in language definition. In *Proceedings of the Companion Publication of the 13th International Conference on Modularity*, MODULARITY '14, pages 1–2, New York, NY, USA. ACM.

Willink, E. D. (2012). An extensible ocl virtual machine and code generator. In *Proceedings of the 12th Workshop on OCL and Textual Modelling*, OCL '12, pages 13–18, New York, NY, USA. ACM.