Proceedings

Ulrich W. Eisenecker
Christian Bucholdt
(Eds.)

SLE 2012

Doctoral Symposium at the 5th International Conference on Software Language Engineering

Dresden, Germany

25 September, 2012

CEUR Publishing Platform

Foreword

This volume contains the proceedings of the Doctoral Symposium at the 5th International Conference on Software Language Engineering, 25th of September 2012, hosted by the Faculty of Information Science of the Technical University of Dresden, Germany. Previous editions were held in Braga, Portugal (2011), Eindhoven, Netherlands (2010), Colorado, USA (2009) and Toulouse, France (2008). The International Conference on Software Language Engineering (SLE) aims to bring together the different sub-communities of the software-language-engineering community to foster cross-fertilisation and to strengthen research overall. Within this context the Doctoral Symposium at SLE 2012 contributes towards these goals by providing a forum for both early and late-stage PhD students to present their research and get detailed feedback and advice from researchers both in and out of their particular research area.

The Program Committee of the Doctoral Symposium at SLE 2012 received 10 submissions. We would like to thank all authors for submitting their papers. Each paper was reviewed by at least three reviewers. Based on the review reports and intensive discussions conducted electronically, the Program Committee selected 8 regular papers.We would like to thank the Program Committee members and all reviewers for their efforts in the selection process.

In addition to contributed papers, the conference program includes a keynote. We are grateful to Dr. Steffen Greiffenberg, University of Cottbus, Germany, for accepting our invitation to address the symposium.

We also would like to thank the members of the Steering Committee, the Organising Committee as well as all the other people whose efforts contributed to make the symposium a success.

The support of our industrial sponsors is essential for SLE 2012. We cordially express our gratitude to (in alphabetical order)

- ACM SIGPLAN
- ANECON GmbH
- DevBoost GmbH
- Elsevier B.V.
- ISX Software GmbH
- Springer LNCS

Furthermore, we thank our academic sponsors, European Association for Programming Languages and Systems and Technical University of Dresden, Germany and the Software Technology Group at the Technical University of Dresden, Germany.

Ulrich W. Eisenecker
Co-Chair
University of Leipzig, Germany

Christian Bucholdt
Co-Chair
Plauen, Germany

Doctoral Symposium at SLE 2012 Organization

Program-Co-Chairs:	Ulrich W. Eisenecker (*University of Leipzig, Germany*)
	Christian Bucholdt (*Plauen, Germany*)
Local Organization:	Birgith Demuth (*Technical University of Dresden, Germany*)
	Sven Karol (*Technical University of Dresden, Germany*)
Program Committee:	Steffen Becker (*University of Paderborn, Germany*)
	David Benavides (*University of Seville, France*)
	Mark van den Brand (*University of Technology Eindhoven, Netherlands*)
	Sebastian Günther (*Vrije University Brussels, Belgium*)
	Michael Haupt (*Oracle, Germany*)
	Arnaud Hubaux (*PReCISE, University of Namur, Belgium*)
	Jaako Järvi (*Texas A & M University, USA*)
	Christian Kästner (*Philipps University Marburg, Germany*)
	Jörg Liebig (*University of Passau, Germany*)
	Roberto Lopez Herrejon (*Johannes Kepler University of Linz, Austria*)
	Johannes Müller (*University of Leipzig, Germanyl*)
	Oscar Nierstraszr (*University of Bern, Switzerland*)
	Zoltan Porkolab (*Eötvös Loránd University, Hungary*)
	Jaroslav Poruban (*University of Kosice, Slovakia*)
	Rick Rabiser (*Johannes Kepler University of Linz, Austria*)
	Gunther Saake (*University of Magdeburg, Germany*)
	Michal Valentai (*University of Technology Prague, Czech Republic*)
	Valentino Vranic (*University of Technology Bratislava, Slovakia*)
	Heike Wehrheim (*University of Paderborn, Germany*)

Table of Contents

Methodology as Theories in Business Informatics
[Keynote Abstract]

Dr. Steffen Greiffenberg

semture GmbH
Dresden, Germany
steffen.greiffenberg@semture.de

Abstract

Since its existence business informatics endeavors to establish itself as a science and to create unique characteristics towards pure computer science. In this keynote theory requirements are outlined and proved as necessities for a science. Furthermore, methods for the development of business information systems as possibilities for theories in business informatics are proposed.
The explication of study designs within business informatics is currently hardly practiced. Thereby, problems regarding objective, replicability and validity of research findings may occur. Those can reflect in the following questions: What is the purpose of this model? Why does the reference model looks just so? What is the aspiration of this model and how can it be verified?
This keynote presumes that the reason for this insufficient explication is an inadequate assistance for the researchers task. Thus, the keynotes target is to draft a method for a concept of study designs in conceptual modeling research. Combined with this method is the hope that the researcher will be equipped with the skills to facilitate the explication of a study design.

Bio

Dr. Steffen Greiffenberg is a visiting professor at the Technical University of Cottbus and managing partner of the semture GmbH in Dresden. The semture GmbH is building software modeling products.

Interoperability of Software Engineering Metamodel: Lessons Learned

Muhammad Atif Qureshi

School of Software, Faculty of Engineering and IT, University of Technology, Sydney, Australia

Abstract. Use of models and modelling languages in software engineering is very common nowadays. To formalize these modelling languages, many metamodels have been proposed in the software engineering literature as well as by standard organizations. Interoperability of these metamodels has emerged as a key concern for their practical usage. We have developed a framework for facilitating metamodel interoperability based on schema matching and ontology matching techniques. In this paper we discuss not the techniques used but rather we focus on the lessons we have learned by applying the framework on several pairs of metamodels for finding similarities between them. We have highlighted some areas where these techniques can be beneficial and also pointed out some limitations of these techniques in this domain.

1 Problem Description and Motivation

Many metamodels have been proposed in different domains of software engineering such as process [1], product [2], metrics [3] and programming [4]. Most of these metamodels have been developed independently of each other with shared concepts being only accidental. These metamodels are evolving continuously and many versions of these metamodels have been introduced over the years. This evolution has extended not only the scope but their size [5] and complexity as well. The need to formulate a way in which these metamodels can be used in an interoperable fashion has emerged as a key issue in the practical usage of these metamodels. There are several benefits of such interoperability including: reduced joint complexity, ease of understanding and use for newcomers, portability of models across modelling tools and better communication between researchers [6]. This overall need is also emphasized by the software engineering community [7] and further endorsed by the rise of industry interest as well as various conferences and workshops on the topic [8]. To have interoperability between any pair of metamodels, similarities between the elements of metamodels need to be identified. This is undertaken by a matching technique as yet little utilized for metamodels although widely used in ontology engineering. Close similarity between metamodels and ontologies [7],[9],[10] suggests that it should be efficacious to adopt ontology matching techniques for facilitating meta-model interoperability with a first step of linguistic matching. Indeed, ontologies are also helpful in reducing semantic ambiguity [9], helping not only to improve the semantics of a metamodel [10] but also providing a potential way in which these meta-models can be bridged with each other to be interoperable. A framework [11] for facilitating interoperability of metamodels has been developed based on the ontology merging and schema matching techniques. The frame-work was applied to several pairs of metamodels including OSM [12], BPMN [13], SPEM [1] and some multi agent systems (MAS) metamodels. In this paper we discuss the lessons learned by applying the framework on these metamodels. We have highlighted the areas of metamodel interoperability that can be assisted by using these techniques as well as discussing some of their limitations. In Section 2 we briefly present our framework for metamodel interoperability. Section 3 presents the lessons learned during the application of this framework to several metamodels, followed by a conclusion and summary of likely future work (Section 4).

2 Proposed Solution

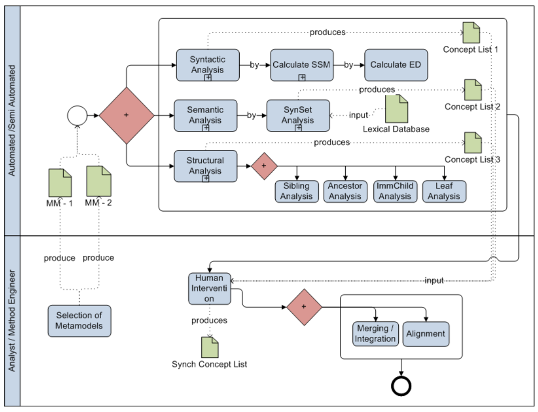


Fig. 1. Metamodel Interoperability Framework [11]

The framework for metamodel interoperability is depicted in Fig. 1 as a BPMN diagram. The framework has two major activities: Linguistic Analysis and Ontological Analysis. These are further divided into subactivities, as represented in the digram. While trying to make metamodels interoperable using this framework, we assume that there exists some commonality between a pair of metamodels. It is necessary to identify the potential common concepts (conceptual elements) that can be shared between two metamodels. The detailed discussion on this framework is not our focus in this paper but can be found in [11]. The overall similarity of any pair of elements is based on the three different types of similarities among them: syntactic, semantic and structural. In applying the framework to a variety of metamodels, several thousand different permutations were computed for the comparison of the metamodel elements. The following sections elaborate our experience of using this framework and discuss the lessons we have learned during the experiment.

3 Lessons Learned: Limitations and Opportunities

3.1 Syntactic Matching

Opportunities: Syntactic matching between a pair of metamodels is based on a comparison between the names of the conceptual elements within those metamodels. Different techniques in the literature are available that can be used for such comparison. One such technique is known as string-edit distance of simply edit distance (ED) [14], which counts the number of token insertions, deletions and substitutions needed to transform one lexical string S1 to another S2, viewing each string as a list of tokens. For example the value of ED for two strings Brian and Brain is 2. Various other techniques for string comparison are used in different domains e.g. N-gram, Morphological Analysis (Stemming), and Stop-Word Elimination. ED can be used then to calculate the syntactic similarity (SSM) between a pair of elements [15]. Lessons Learned: These techniques can be useful in comparing the elements with-in the same domain e.g. domain ontologies; where elements with the same name have (most of the times) the same meaning. The problem with these techniques in the context of metamodels is that they are not effective when applied standalone. Our experience with metamodel matching shows that considering only syntactic similarity measures, isolated from their semantics, creates misunderstanding by expressing the same meanings in different terms. For example confirmation and notification has approximately 60

3.2 Matching the Semantics

Metamodels are generally treated as a model of a modelling language [16], [17],[18][19]. These modelling languages are designed (mostly) for specific domains. Therefore, we believe that to compare the semantic similarity of metamodel elements, it is important to consider both perspectives: linguistic and ontological. The linguistic semantics involves checking the semantics of the metamodel elements from that modelling languages perspective e.g. their properties (attributes), types of attributes and to some extent their behaviour as well. On the other hand, on-tological semantics means finding the elements that have the same meaning but may have been presented with different names. Opportunities: Techniques for comparing class diagrams e.g. [20],[21] can be utilized to find the similarities between metamodel elements, especially for the metamodels that are represented using object-oriented classes (meta-classes) e.g. OMGs family of meta-models. Different approaches in the area of computational linguistics and natural language processing can be used to find ontological semantic similarity e.g. finding the synonyms of a given conceptual element of one metamodel and looking for those synonyms in the second metamodel. Synonyms can be found using any lexical data-base e.g. a dictionary. WordNet [21] is one lexical database that can be used for finding synonyms and word senses. WordNet is a registered trademark for Princeton University and contains more than 118,000 word forms and 90,000 different word senses. Lessons Learned: We have observed that finding ontological semantic similarity is very important as there are so many such conceptual elements in metamodels presented with different names. For example, Person in OSM [12] can be semantically matched with the Human Performer in BPMN [13]; although both have low syntactic similarity. Beside synonyms, hyponyms (sub-name, e.g. sugar-maple and maple), hypernyms (supername, e.g. step and footstep) can also be used to find semantic relevant elements, but none of these are considered so far in any technique. Similarly, meronyms (part-name, e.g. ship and fleet) and holonyms (whole-name, e.g. face and eye) can also be useful to find these similarities. Another problem is how to combine both linguistic and ontological semantic similarity for a pair of conceptual elements. Which one of them is more important and how much weight should be assigned to each of them is still unaddressed.

3.3 Comparing the Structures

Besides their level of abstraction, a metamodel is treated as a (conceptual) model of a language [22]. For a good similarity comparison between any pair of conceptual models, not only their syntax and semantics but also their structure should be compared. Opportunities: Different techniques have been proposed in the literature for structural similarity of conceptual models. Some of these [22], [14] compare the structure of business process models, whilst others [23],[24] are for matching the structure of conceptual models based on graph theory. An alternative to a graph matching technique is the schema matching techniques [24], [25][26][27][28]. In this technique, the structural similarity of two conceptual elements C1 and C2 is calculated based on their structural neighbours - ancestors, siblings, immediateChilds, and leafs. These partial similarities are then calculated by mean values of all the neighbouring concepts. Lessons Learned: The techniques used to compare the structure of business process models (e.g. [22], [14]) cannot be generalized for metamodels as business process models are behavioural models while metamodels represents the structural aspect. Converting the conceptual models to graphs [23], [24] and then applying graph matching algorithms to find the structural similarity between them is not a trivial task. To apply such a graph matching technique, we have to be very careful in the conversion of a class diagram into a graph. True replacement of relationships among classes (e.g. association, generalization, aggregation, composition) into relationships among nodes of a graph (e.g. directed/undirected, weighted/unweighted) is not straightforward. Another barrier for the application of such techniques is that most of the metamodels in the software engineering literature are specified using diagrams, tables and textual explanation. Having a single class diagram for such a huge metamodel is not easy. Techniques based on the planar graph theory like [24] are also not feasible for meta-models because of the basic principle of planar graphs (having no cross edges). Meta-models with a rich set of constructs (classes) like UML can easily violate this rule as it is very difficult to convert class diagrams of these metamodels to graphs without any cross edges. The complexity of these graph matching techniques, as also mentioned by some authors [14], is another barrier to their application in the domain of metamodels, hence making it difficult to apply in practice. Based on the experience of applying these techniques to metamodels, we recommend that we dont need to compare the leaves of any conceptual element in a meta-model. Comparing leaf classes of a given class (conceptual element) only results in low similarity. Also, we think that rather than comparing all the ancestors of a conceptual element, it is better to compare only parent classes of that element.

3.4 Automation

Considering the size and complexity of metamodels [5], it is very convenient to have tool support for matching the similarity of metamodels. Hence, our experience with the matching of metamodels shows that, beside partial tool support, complete automated metamodel matching is not possible. Opportunities: Automation in syntactic matching of metamodels elements can be achieved by implementing ED (Edit Distance) and SSM (Syntactic Similarity Measure) algorithms using available online calculators for ED and APIs. The ontological semantics of metamodel elements can be matched automatically using lexical databases like WordNet, MS Office Thesaurus and other APIs available. Lessons Learned: Complete automation for metamodel similarity matching, especially for structural similarity, requires well formed formal definitions of metamodels that can be used as an input for any automated tool. Unfortunately, besides XML definitions for some of the metamodels (OMG metamodels with XMI definitions), metamodels lacks a formal specification and are mostly specified using a combination of textual descriptions, tables and class diagrams. Another important barrier in the complete automation is that coefficients in the equations we used do not have any fixed values and have to have value assigned by the domain expert at the time of the matching. Also, the ontological semantic similarity analysis requires the experts intellectual input to decide whether two conceptual elements are equal or not.

3.5 Refactoring

Lessons Learned: Most of the metamodels have two orthogonal forms of conceptual elements: linguistic and ontological (as also highlighted by [17]). The former represent the language definition while the latter describe what concepts exist in a certain domain and what properties they have. These two types of elements are mingled with each other in most of the metamodels and there is no explicit boundary between them. An important consideration regarding metamodel matching is to separate these two types of elements; we call it refactoring.

Metamodels need to be first refactored before matching can occur. This refactoring is required to remove the conceptual elements in metamodels that are not related to the domain of interest. Rather, most of these elements are linguistic and are present in order to maintain (glue) the structure of metamodels. For example, Resource Parameter Binding and Parallel Gateway in BPMN [13] are the concepts that are related to the language definition of BPMN and are not worth matching with any other metamodel of the same domain since every metamodel has its own language definitional elements. Rather, it is better to match the conceptual elements that are related to the domain of interest e.g. matching Activity in BPMN [13] with Activity in SPEM [1], which are more related to the common domain of interest: Workflows and Processes.

3.6 Ontology Oriented Metamodels

Lessons Learned: Our experience of matching metamodels showed that there is a high heterogenity between the ontological elements of metamodels. However, it has been observed that a major reason for that heterogeneity is the lack of a common ontology or taxonomy. Much better results in interoperability of metamodels can be achieved if metamodels share some common ontology or taxonomy of the domain of interest; as also highlighted by [8]. The use of a common ontology for designing/redesigning metamodels can result in better interoperability. For example, the use of the UFO (Unified Foundation Ontology) to redesign UML [29]. Metamodels based on a common ontology will reduce the differences of similarity matchings, especially in syntactic and ontological semantics matching.

4 Conclusion

In this paper we have discussed some of the limitations and opportunities in the field of metamodel interoperability. These recommendations are based on the application of a framework that we have developed and applied on several metamodels to find their similarities. We have come to conclude that, for better similarity findings, not only the syntax but also the semantics and structure of metamodel elements should be matched. Metamodels needs to be refactored to separate out the ontological elements before matching for more pragmatic results. To avoid the problems of syntactic and semantic ambiguities between the elements, we recommend that metamodels should be based (or at least utilize) upon some common domain ontology. Also we have shown that complete automation of matching metamodel elements is not possible and does require substantial human intervention.

References

1. OMG: Software and systems process engineering meta-model specification (2008)
2. OMG: Unified modeling language (2009)

3. OMG: Architecture-driven modernization (adm): Software metrics meta-model (smm) (2009)
4. Azaiez, S., Huget, M.P., Oquendo, F.: An approach for multi-agent metamodelling. Multiagent and Grid Systems **2**(4) (2006) 435–454
5. Henderson-Sellers, B., Qureshi, M.A., Gonzalez-Perez, C.: Towards an interoperable metamodel suite: Size assessment as one input. International Journal of Software and Informatics **6 (2)**(2) (2012)
6. Beydoun, G., Low, G., Henderson-Sellers, B., Mouratidis, H., Gomez-Sanz, J.J., Pavon, J., Gonzalez-Perez, C.: Faml: A generic metamodel for mas development. IEEE Trans. Softw. Eng. **35**(6) (2009) 841–863
7. Henderson-Sellers, B.: Bridging metamodels and ontologies in software engineering. Software and Systems **84**(2) (2011) in press
8. Bézivin, J., Soley, R.M., Vallecillo, A.: Proceedings of the first international workshop on model-driven interoperability (2010)
9. Tran, Q.N.N., Low, G.: Mobmas: A methodology for ontology-based multi-agent systems development. Inf. Software Technol **50**(7-8) (2008) 697–722
10. Devedzić, V.: Understanding ontological engineering. Communications of the ACM **45**(4) (2002) 136–144
11. Qureshi, M.A.: Interoperability of software engineering metamodels (2012)
12. OMG: Organization structure metamodel (osm) 3rd initial submission (2009)
13. OMG: Business process model and notation (bpmn) ftf beta 1 for version 2.0 (2009)
14. Dumas, M., Garca-Banuelos, L., Dijkman, R.: Similarity search of business process models. IEEE Data Eng. Bull **32**(3) (2009) 23–28
15. Maedche, S.: Comparing ontologies - similarity measures and a comparison study. Technical report, Institute AIFB, University of Karlsruhe, Internal Report (2001)
16. Henderson-Sellers, B., Gonzalez-Perez, C.: An investigation of the validity of strict metamodelling in software engineering. submitted to IEEE Trans. Software Eng. (2011)
17. Atkinson, C., Kuhne, t.: Model-driven development: A metamodeling foundation. IEEE Software **20**(5) (2003) 36–41
18. Gašević, D., Kaviani, N., Hatala, M. In: On Metamodeling in Megamodels. Volume 4735/2007. Speinger (2007) 91–105
19. Kuhne, T.: Matters of metamodelling. Software and System Modeling **5**(4) (2006) 395–401
20. Girschick, M.: Difference detection and visualization in uml class diagrams. technical report. Technical report (2006)
21. Miller, G.A.: Wordnet: A lexical database for english. Communications of the ACM **38**(11) (1995) 39–41
22. Ehrig, M., Koschmider, A., Oberweis, A.: Measuring similarity between semantic business process models (2007) 1274465 71-80.
23. Voigt, K., Heinze, T.: Metamodel matching based on planar graph edit distance (2010) 1875866 245-259.
24. Melnik, S., Garcia-Molina, H., Rahm, E.: Similarity flooding: a versatile graph matching algorithm and its application to schema matching (2002 2002)
25. Bernstein, P.A.: Applying model management to classical meta data problems (2003)
26. Chukmol, U., Rifaieh, R., Benharkat, N.A.: Exsmal: Edi/xml semi-automatic schema matching algorithm (2005)

27. Filipe, J., Cordeiro, J., Sousa, J., Lopes, D., Claro, D.B., Abdelouahab, Z. In: A Step Forward in Semi-automatic Metamodel Matching: Algorithms and Tool. Volume 24 of Lecture Notes in Business Information Processing. Springer Berlin Heidelberg (2009) 137–148
28. Lopes, D., Hammoudi, S., de Souza, J., Bontempo, A.: Metamodel matching: Experiments and comparison (Oct. 2006 2006)
29. Guizzardi, G., Wagner, G. In: Using the Unified Foundational Ontology (UFO) as a Foundation for General Conceptual Modeling Languages. Springer-Verlag (2010)

Aspect-Oriented Language Mechanisms for Component Binding

Kardelen Hatun, Christoph Bockisch, and Mehmet Akşit

TRESE, University of Twente
7500AE Enschede
The Netherlands
http://www.utwente.nl/ewi/trese/
{hatunk,c.m.bockisch,aksit}@ewi.utwente.nl

Abstract. Domain Specific Languages (DSLs) are programming languages customized for a problem/solution domain, which allow development of software modules in high-level specifications. Code generation is a common practice for making DSL programs executable: A DSL specification is transformed to a functionally equivalent GPL (general-purpose programing language) representation. Integrating the module generated from a DSL specification to a base system poses a challenge, especially in a case where the DSL and the base system are developed independently. In this paper we describe the problem of integrating domain-specific modules to a system non-intrusively and promote loose coupling between these to allow software evolution. We present our on-going work on aspect-oriented language mechanisms for defining object selectors and object adapters as a solution to this problem.

1 Introduction

Complex systems are created by assembling software components of various types and functions. Reuse is essential and components created for a system are required to continue working after the system has evolved. Some components may be domain-specific, meaning their structure and functionality can be defined using the fundamental concepts of the relevant domains. A domain-specific language (DSL) provides expressive power over a particular domain. It allows software development with high-level specifications; if general-purpose programming languages are used, development may take a considerable programming effort.

The specifications written in a DSL can be processed in various ways. These are comprehensively described in [4] and [3]. Generative programming [2] is one of the processing options and has become highly popular with the emergence of user-friendly language workbenches. Most language workbenches provide a means to develop a compiler for the DSL, facilitating code generation in general-purpose languages. (A comparison matrix for language workbenches can be found in [1].)

In this paper we focus on the integration of components into target systems. "Component" is a very general concept and it can be realized in different forms, depending on the system. We particularly focus on a subset of components, *domain-specific components*, which are instances of domain-specific meta-models. The component structure is described with a DSL and the semantics are embedded into code generation templates, which are used to generate a component that is tailored towards a base system's requirements.

Integrating a generated component into a system poses three main challenges. (1) When adding unforeseen functionality to a system, no explicit *hooks* exist for attaching the generated component. In this case it may be necessary to modify the generated code, the system code or both to make the connection, which will expose the system developer to the implementation details of the generated code. (2) The interfaces of the generated component and the target system should be compatible to work together, which is generally not the case. Then one of the interfaces should be adapted, possibly by modifying the system's or the component's implementation or their type-system. (3) When the component or the target system evolves, the links between them must be re-established.

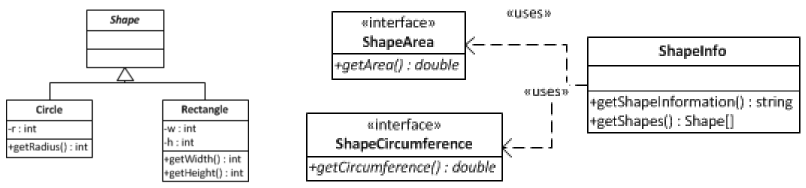
Current aspect-oriented languages offer mechanisms to modularly implement solutions for the first challenge. It can be solved by defining pointcuts that are used as hooks to a system. The second challenge is our main focus. Existing AO-languages offer limited mechanisms for implementing adapters between interfaces. AspectJ inter-type declarations can be used to make system classes to implement appropriate interfaces, however this approach is type-invasive. CaesarJ offers a more declarative approach with *wrappers*, but their instantiation requires pointcut declarations or they should be explicitly instantiated in the base system. The links mentioned in the third challenge are the adapter implementations mentioned in the second challenge and they represent the binding between two components. However current AO languages do not offer a declarative way for describing such a binding; an imperative programming language will lead to less readable and less maintainable implementation, which is fragile against software evolution.

2 Approach

In order to overcome the shortcomings of the existing approaches we intend to design a declarative way of implementing object adapters which is used together with a specialized pointcut for selecting objects. The object adapter pattern is common practice for binding two components that have incompatible interfaces. Our approach is aspect-oriented and it will provide the means to non-intrusively define and instantiate object adapters, inside aspects. These adapters represent links between the component and the system; their declarative design requires a declarative way of selecting the adaptee objects.

In order to select objects to be adapted, we have designed a new pointcut mechanism called *instance pointcut* which selects sets of objects based on the execution history. An instance pointcut definition consists of three parts: an identifier, a type which is the upper bound for all objects in the selected set, and a specification of relevant objects. The specification utilizes *pointcut expressions* to select events that define the begin and end of life-cycle phases and to expose the object. At these events, an object is added or removed from the set representing the instance pointcut. It is possible to access all objects currently selected by an instance pointcut and to be notified, when an object is added or removed. New instance pointcuts can be derived from existing ones in several ways. Firstly, a new instance pointcut can be derived from another one by restricting the type of selected objects. Secondly, a *subset* or a *super-set* of an existing instance pointcut can be declared whereby the specification of the life-cycle phase is either narrowed down or broadened. Finally, instance pointcut declarations can be composed arbitrarily by means of boolean operators.

Adapter declarations refer to the sets selected by instance pointcuts, and automatically instantiate adapters for each object in the referred set. Unlike inter-type declarations, adapter declarations are not type invasive; they are compiled to the object adapter pattern and they do not change the type hierarchy of the contained object. They also do not require explicit instantiations.



(a) The shapes hierarchy (b) ShapeInfo class that requires two unsupported interfaces

Fig. 1: Incompatible interfaces: Shape and ShapeInfo

The header of an adapter declaration consists of an identifier, the list of interfaces the adapter implements and an instance pointcut reference which contains the adaptee objects. In the body of an adapter declaration implementation of the interface methods is provided. In Figure 1a a Shape hierarchy and the interfaces offered by the classes in this hierarchy is shown. The ShapeInfo class uses ShapeArea and ShapeCircumference interfaces to query existing Shapes (Figure 1b). However none of the classes in the shapes hierarchy implements these interfaces, hence they should be adapted. Assume that there is a class called CircleCreator which has two methods: createLargeCircle and createSmallCircle. We can define an instance pointcut called largeCircles which selects the set of Circle objects that are created by the createLargeCircle method. Here instance pointcuts give us expressive power over selecting specific objects as adaptees. Listing 1 shows an example of an adapter declaration. The name of the adapter is CircleAdapter and it implements the interfaces defined in the square brackets; CircleAdapter adapts the objects selected by the circles instance pointcut. In the body of the adapter the implementations of the two declared interfaces are provided. The **adaptee** keyword refers to an object in the circles set.

```
declare adapter: CircleAdapter[ShapeArea, ↙
    ShapeCircumference] adapts largeCircles
{
  public double getArea()
  {
    return Math.pow(adaptee.getRadius(),2)*Math.PI;
  }
  public double getCircumference()
  {
    return 2*adaptee.getRadius()*Math.PI;
  }
}
```

Listing 1: The adapter declaration for Circle objects

3 Compilation and Run-time Support

In our prototype implementation instance pointcuts are compiled to AspectJ and Java code. Roughly, an instance pointcut is transformed to several AspectJ pointcuts, advice declarations, a set structure and methods for altering this set. Adapter declarations will also be compiled to AspectJ. According to our initial analysis an adapter declaration will map to a Java class for the adapter and advice bodies for initializing adapters. These advice bodies will reference the pointcuts generated from the instance pointcut which is referenced by the adapter declaration.

We intend to provide run-time support for retrieving adapter instances. Adapters are automatically initialized when an adaptee object satisfying the referenced instance pointcut's conditions become available. These adapter instances can be indexed and accessed through a run-time library. To do this, we have the requirement that the results of a retrieval will always be non-ambiguous e.g. if a query to retrieve a single adapter instance, matches two adapters, then there should be appropriate resolution mechanisms or user feedback to overcome the issue.

References

1. Language workbench competition comparison matrix (2011), `www.languageworkbenches.net`
2. Czarnecki, K.: Overview of generative software development. In: Unconventional Programming Paradigms, Lecture Notes in Computer Science, vol. 3566, pp. 97–97. Springer Berlin , Heidelberg (2005)
3. Fowler, M., Parsons, R.: Domain-specific languages. Addison-Wesley (2010)
4. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. 37, 316–344 (December 2005)

Supporting Visual Editors using Reference Attributed Grammars

Niklas Fors

Department of Computer Science, Lund University, Lund, Sweden
`niklas.fors@cs.lth.se`

Abstract. Reference attributed grammars (RAGs) extend Knuth's attribute grammars with references. These references can be used to extend the abstract syntax tree to a graph. We investigate how RAGs can be used for implementing tools for visual languages. Programs in those languages can often be expressed as graphs.

1 Introduction

For certain applications domain-specific languages (DSLs) play an important role, since they can make it easier to solve problems in the domain in a more concise way than a general purpose language (GPL) [1]. DSLs are especially advantageous when the language user is a domain expert and not a programmer. Other advantages of DSLs over GPLs are easier domain analysis and optimizations [2]. However, there are several difficulties with DSLs as well. Developing DSLs often require compiler implementation knowledge, which is not necessary when creating a library. It is also desirable that the domain is well-understood, where abstractions are more likely to remain stable. Limited resources for creating and maintaining a DSL can be a problem, especially for small communities.

A DSL user may want different tools, such as a batch compiler and an integrated development environment (IDE). Such tools often share analysis, and reusing specification between them can reduce the implementation effort and help to keep the tools consistent with each other. One tool aiming for reuse of language and compiler specification is JastAdd [3], which we use in our research.

Often, new insights and knowledge about a particular domain arise over time, leading to new abstractions in the DSL. In such cases, the language developer typically wishes to extend the existing DSL in a backward compatible way. JastAdd has proven to be useful for extending languages; compilers for complex languages like Java and Modelica have successfully been implemented and extended with new language constructs [4,5].

In several domains, a visual notation is preferable to a textual notation, which can increase the accessibility for non-programmers. Such languages are called domain-specific visual languages (DSVLs). Some languages, such as Modelica [6], support both a textual and a visual syntax. This allows the software engineer and the domain expert to use the most convenient notation.

We want to investigate how the metacompilation system JastAdd, and its formalism reference attributed grammars [7], can be used for implementing tools for visual languages in general, and DSVLs in particular.

2 Background

There are several ways for defining the semantics, that is the meaning, of a programming language. Examples include denotational semantics [8,9], operational semantics [10,11] and attribute grammars [12].

In this work, we will use reference attributed grammars (RAGs), which is an extension to Knuth's attribute grammars [12]. RAGs allow the value of an attribute to be a reference to a node in the abstract syntax tree. For example, an identifier usage may have a reference attribute that refers to its declaration. The usage node can use this reference attribute to access attributes from the declaration node, for example, to obtain the type of the declaration. Using reference attributes, a graph can be superimposed on the AST, which can be used to model graph-based languages. Since graphs can be cyclic, the support of circular reference attributes [13] in JastAdd can be helpful, which solve circular attribute definitions using fixed-point iteration. Earlier work has shown that circular reference attributes are practical and provide a concise definition for the implementation of control-flow and data-flow for Java programs [14]. Other attribute grammar system supporting reference attributes include Silver [15] and Kiama [16].

For visual languages, there are several ways to define the syntax of a language. There is, however, no consensus on what formalism to use, in contrast to textual languages, where Chomsky's context-free grammars are widely used. Examples of formalisms are graph grammars [17,18,19], metamodeling [20,21] and constraint multiset grammars [22].

Erwig [23] proposed a separation of the concrete and abstract syntax for a visual language, and defined the semantics on the latter. The abstract syntax was modeled as a directed labeled multi-graph (there can be several edges between two nodes). Our approach differs from Erwig in that the base structure is the AST compared to a graph, and instead we use reference attributes to extend the AST to a graph. We also use RAGs to define the semantics. We think that a tree is practical when a textual syntax is desired, since it is straightforward to serialize and easy to create new concrete syntaxes. Rekers et al. [19] also suggest a separation between the concrete syntax and abstract syntax for visual languages.

Another related work is reusable visual patterns by Schmidt et al. [24], implemented with attribute grammars, but without use of RAGs. These patterns are predefined, and examples are lists, tables, sets, lines etc. The language developer can choose among the patterns and use the ones that matches the visual language, and can customize visual properties for the patterns. If a pattern is missing, a new can be added by defining attribute equations.

3 Research Proposal

We want to investigate how well suited RAGs are for defining the semantics for a visual language, and how to use RAGs for semantics-dependent editor interaction. In particular, we have the following research goals:

– Declarative semantic specification.
– Reuse of semantic specifiation.
– Semantic support in visual editor, for example, showing computed properties.
– Support languages with both a textual and visual syntax.
– Automatic incremental evaluation, using work by Söderberg et al. [25].
– Integration with other editor frameworks, such as JastGen [26].

3.1 Challenges

We have identified the following research challenges, based on experience from the preliminary work described below.

Defining visual syntax. When defining a visual language, there should be a way to define its syntax and the visual representation, that is, properties such as shape and color. One way is to have a DSL supporting that and which includes concepts such as nodes, connections and containment. To specify how nodes can connect, constraints can be used. Attributes are one way to specify such constraints. Another way is to use OCL as Bottoni et al. [20] have done or create a predicate language as Esser et al. [27]. It should also be possible to show attribute values. This can be used to show computed properties such as cycles in graphs, inferred types, etc.

Another approach for defining the syntax is by graph grammars [19], such as hypergraphs [17], where the visual representation is transformed to a graph, which is then parsed. One benefit of graph grammars is that they allow both free-hand and syntax-directed editing. However, earlier work [28] indicates that metamodeling using GMF [21] is easier to use but less expressive than hypergraphs, which is something we have experienced as well. We think it is desirable that the specification format is easy to use, so it is appealing for a software engineer.

We want to find and choose one approach and connect it to RAGs.

Model consistency. In the visual editor, we have both the view and the AST model, and a challenge is to keep these models consistent with each other. One way is to let the visual editor update the AST, which then informs the view about the change and is updated accordingly. If the editor supports both textual and visual editing, is this technique feasible? Another way is to support bidirectional transformations [29].

Automatic layout. We want to find out how to handle layout that can be sensitive to semantics. A language from ABB has this property, as described below. For these languages, automatic layout must take this semantics into consideration, in order to not inadvertently change the meaning of a diagram.

Erroneous diagrams. It can be helpful for the language user if it is possible to temporarily have diagrams that contain errors. For example, when two edges are being switched and it is forbidden to have more than one edge to a node, then it may be useful to temporarily have one edge pointing to nothing.

Undo / redo functionality. To increase the usability of editors for visual languages, functionality like undo and redo is often desirable. We would like to generate edit operations for the AST that automatically provide support for undo and redo functionality.

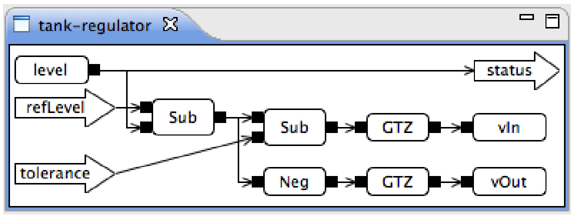
Refactoring. Another challenge is to support refactoring in visual languages with RAGs. Is it possible to extend the work done by Schäfer et al. [30]?

3.2 Methodology

The research methodology we use is similar to design science research in information systems. Instead of trying to understand the reality, as natural science does, design science tries to create new innovative artifacts that have utility [31]. We develop prototypes that are based on practical applications, as the preliminary work with ABB demonstrates. The research contribution is the technical artifact itself and the development of the foundations of RAGs. Design science consists of two activities, *build* and *evaluate*, and is an iterative process; the artifact is continuously evaluated to provide feedback to the building activity [32]. The evaluation will include performance measurements, for example, the response time for standard editing operations, as well as evaluation of specification size and modularity. We develop a prototype visual language together with ABB using RAGs, and it would be interesting to see if the engineers at ABB successfully can experiment with new language constructs for this language in a modular way.

3.3 Preliminary Work

We have started a collaboration with ABB, and designed a control language based on a industrial language from ABB [33]. This language has both a textual and a visual syntax. A diagram in the language contains blocks that are executed periodically and connections between them that specify the data flow. One interesting aspect of the language is that the semantics are influenced by the visual placements of the blocks. The execution order of the blocks is primarily defined by the data flow, and secondary by visual placements. To define the semantics without visual coordinates, the visual placements are reflected in the declaration order of the blocks in the textual syntax. We have implemented a visual editor for this language using RAGs and the Graphical Editing Framework (GEF), an Eclipse-based framework. A screenshot of the editor can be seen in Fig. 1, with the corresponding textual syntax.



(a) Visual syntax

```
diagramtype tank-regulator(
    Int refLevel, Int tolerance
    => StatusData status) {
  StatusMonitor level;
  Sub Sub_1;
  ...
  connect(refLevel, Sub_1.in1);
  ...
}
```

(b) Textual syntax

Fig. 1: A model for regulating the liquid volume in a tank, using two valves, `vIn` and `vOut`.

4 Conclusions

We have in this paper described the opportunity to use RAGs for implementing tools for visual languages. Attribute gammars is a proven and convenient formalism to specify the semantics of a language. RAGs extends AGs with references, and makes it possibly to superimpose graphs on an AST, which can be used to represent programs of visual languages. Expected benefits of using RAGs include reuse of compiler specification between different tools and the possibility to experiment with new language constructs in a modular way. The expected contributions are the prototype system and development of the foundations of RAGs to support visual languages. Research goals have been described and research challenges identified. Finally, we mention design science research as our research methodology and describe the preliminary work with ABB that serves as a case study.

Acknowledgements

We thank the anonymous reviewers for interesting and useful comments on a earlier draft of this paper.

References

1. Deursen, A., Klint, P.: Little languages: Little maintenance? Journal of Software Maintenance: Research and Practice **10**(2) (1998) 75–92
2. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. **37**(4) (2005) 316–344
3. Ekman, T., Hedin, G.: The JastAdd system - modular extensible compiler construction. Science of Computer Programming **69**(1-3) (2007) 14–26
4. Ekman, T., Hedin, G.: The Jastadd Extensible Java Compiler. In: OOPSLA 2007, ACM (2007) 1–18
5. Hedin, G., Åkesson, J., Ekman, T.: Extending languages by leveraging compilers: from Modelica to Optimica. IEEE Software **28**(3) (March 2010) 68–74

6. Modelica Association: Modelica - A Unified Object-Oriented Language for Systems Modeling - Language Specification Version 3.3. (2012) Available from: `http://www.modelica.org`.
7. Hedin, G.: Reference Attributed Grammars. In: Informatica (Slovenia). 24(3) (2000) 301–317
8. Scott, D.: Outline of a mathematical theory of computation. Oxford University Computing Laboratory, Programming Research Group (1970)
9. Scott, D., Strachey, C.: Toward a mathematical semantics for computer languages. Oxford University Computing Laboratory, Programming Research Group (1971)
10. Kahn, G.: Natural semantics. In: 4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87. Volume 247 of LNCS., Passau, Germany, Springer (1987) 22–39
11. Plotkin, G.D.: A Structural Approach to Operational Semantics. Technical report, Department of Computer Science, University of Aarhus (1981)
12. Knuth, D.E.: Semantics of Context-free Languages. Math. Sys. Theory **2**(2) (1968) 127–145 Correction: *Math. Sys. Theory* 5(1):95–96, 1971.
13. Magnusson, E., Hedin, G.: Circular Reference Attributed Grammars - Their Evaluation and Applications. Science of Computer Programming **68**(1) (2007) 21–37
14. Söderberg, E., Ekman, T., Hedin, G., Magnusson, E.: Extensible intraprocedural flow analysis at the abstract syntax tree level. Science of Computer Programming (2012)
15. Wyk, E.V., Bodin, D., Gao, J., Krishnan, L.: Silver: An extensible attribute grammar system. Science of Computer Programming **75**(1-2) (2010) 39–54
16. Sloane, A.M., Kats, L.C.L., Visser, E.: A pure object-oriented embedding of attribute grammars. Electr. Notes Theor. Comput. Sci. **253**(7) (2010) 205–219
17. Minas, M., Viehstaedt, G.: DiaGen: A generator for diagram editors providing direct manipulation and execution of diagrams. In: Proceedings of the IEEE International Symposium on Visual Languages, IEEE (1995) 203–210
18. Ehrig, K., Ermel, C., Hänsgen, S., Taentzer, G.: Generation of visual editors as Eclipse plug-ins. In: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. ASE '05, ACM (2005) 134–143
19. Rekers, J., Schürr, A.: Defining and parsing visual languages with layered graph grammars. Journal of Visual Languages and Computing **8**(1) (1997) 27–55
20. Bottoni, P., Costagliola, G.: On the definition of visual languages and their editors. Diagrammatic Representation and Inference (2002) 337–396
21. GMF: The Eclipse Graphical Modeling Framework http://www.eclipse.org/modeling/gmp/
22. Marriott, K.: Constraint multiset grammars. In: Visual Languages, 1994. Proceedings., IEEE Symposium on, IEEE (1994) 118–125
23. Erwig, M.: Abstract syntax and semantics of visual languages. J. Vis. Lang. Comput. **9**(5) (1998) 461–483
24. Schmidt, C., Kastens, U.: Implementation of visual languages using pattern-based specifications. Software: Practice and Experience **33**(15) (2003) 1471–1505
25. Söderberg, E., Hedin, G.: Incremental evaluation of reference attribute grammars using dynamic dependency tracking. Technical Report 98, Lund University (April 2012) LU-CS-TR:2012-249, ISSN 1404-1200.
26. Söderberg, E., Hedin, G.: Building semantic editors using jastadd. In: Proceedings of the of the 11th Workshop on Language Descriptions, Tools and Applications, LDTA 2011, ACM (2011)

27. Esser, R., Janneck, J.: A framework for defining domain-specific visual languages. Workshop on Domain Specific Visual Languages, ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA-2001) (2001)
28. Tveit, M.: Specification of graphical representations-using hypergraphs or meta-models? Norsk informatikkonferanse NIK (2008) 39–50
29. Antkiewicz, M., Czarnecki, K.: Design space of heterogeneous synchronization. In: Generative and Transformational Techniques in Software Engineering II. LNCS, Springer (2008) 3–46
30. Schäfer, M., Ekman, T., de Moor, O.: Sound and Extensible Renaming for Java. In Kiczales, G., ed.: 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008), ACM Press (2008)
31. March, S.T., Smith, G.F.: Design and natural science research on information technology. Decis. Support Syst. **15**(4) (December 1995) 251–266
32. Hevner, A.R., March, S.T., Park, J., Ram, S.: Design science in information systems research. MIS Q. **28**(1) (March 2004) 75–105
33. Fors, N., Hedin, G.: Handling of layout-sensitive semantics in a visual control language. In: Proceedings of the 2012 IEEE Symposium on Visual Languages and Human-Centric Computing, IEEE (2012) poster paper to appear.

Towards a Framework for the Integration of Modeling Languages

Svetlana Arifulina

Department of Computer Science, University of Paderborn, Germany[**]
`svetlana.arifulina@uni-paderborn.de`
`http://is.uni-paderborn.de`

Abstract. In software markets of the future, customer-specific software will be developed on demand from distributed software and hardware services available on world-wide markets. Having a request, services have to be automatically discovered and composed. For that purpose, services have to be matched based on their specifications. For the accurate matching, services have to be described comprehensively that requires the integration of different domain-specific languages (DSLs) used for functional, non-functional, and infrastructural properties. Since different service providers use plenty of language dialects to model the same service property, their integration is needed for the matching. In this paper, we propose a framework for integration of DSLs. It is based on a parameterized abstract core language that integrates key concepts needed to describe a service. Parts of the core language can be substituted with concrete DSLs. Thus, the framework serves as a basis for the comprehensive specification and automatic matching of services.

Keywords: Integration of DSLs, Metamodeling, Service Specifications

1 On-The-Fly Computing

In the Collaborative Research Centre 901 "On-The-Fly Computing" (OTF Computing), we develop techniques to *automatically* configure IT services distributed on world-wide markets in an ad hoc manner to fulfill customer-specific requests[1]. On receiving a request, suitable software and hardware services have to be automatically discovered and composed. For that purpose, services descriptions have to be compared with search requests (*matching*), in order to determine whether they fit. After a service composition has been created, its quality has to be analyzed, in order to determine the most suitable solution to the request.

In the remainder of this paper, research questions in OTF Computing are presented in Section 2. Then, preliminary results on the framework and its contributions are described in Section 3. Finally, Section 4 summarizes the proposed method and contains further steps.

[**] This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901)

[1] For more information refer to `http://sfb901.uni-paderborn.de`

2 Research Challenges and Issues

One of the research challenges of OTF Computing is the automatic configuration of services. It is based on an appropriate matching mechanism and a quality analysis of composed solutions. The appropriate matching mechanism is based on service specifications that, on the one hand, contain all service properties necessary for their accurate comparison and, on the other hand, not too complicated so that they would hamper the efficiency of this comparison. The quality analysis requires service specifications that can be used to derive properties of a composed service, in order to check whether it functions properly. So, a research issue is a service description that enables the automatic and efficient configuration and quality analysis.

One broadly used language to describe web services is Web Service Description Language (WSDL) [CCMW01]. WSDL allows the description of the service in the form of its operation signatures, but does not provide any expressive means to describe the behavior of the service. Another challenge is that in addition to structure and behavior, non-functional properties and infrastructure information have to be specified as well, in order to enable an accurate matching of service level agreements and execution environments. Approaches like WS-* specifications[2] propose solutions for single properties of services but these is still a lack of an integrated language. So, a research issue is an integration of DSLs for different service properties in a consistent comprehensive specification.

One more research challenge of OTF Computing is the fact that service providers use different DSLs to describe the same service properties. Since it is infeasible to enforce the usage of only one language for all service providers and requesters, the flexibility to use their own DSLs must be preserved. However, these heterogeneous service descriptions still have to be matched. Therefore, different mutual exclusive DSLs used for the same service property have to be integrated as well. This kind of integration is often solved in the literature by creating a syntactical mapping between two DSLs that results in merely syntactical matching. So, a research issue is a semantical integration of DSLs that will also yield a matching mechanism considering semantics.

3 Preliminary Results and Contributions

In this section, we introduce a framework for the integration of modeling languages that addresses the research issues introduced in Section 2. An overview of the framework is shown in Fig. 1. The framework consists of an abstract *core language* that serves as a common basis for the integration of DSLs for a comprehensive specification and for matching. The core language accumulates different service properties necessary for the automatic configuration and quality analysis. It is represented by a metamodel, which in turn comprises further metamodels each modeling a certain service property. Furthermore, the core language is *parameterized*, i.e., its parts can be substituted by concrete DSLs.

[2] For more information refer to `http://www.w3.org/2002/ws/`

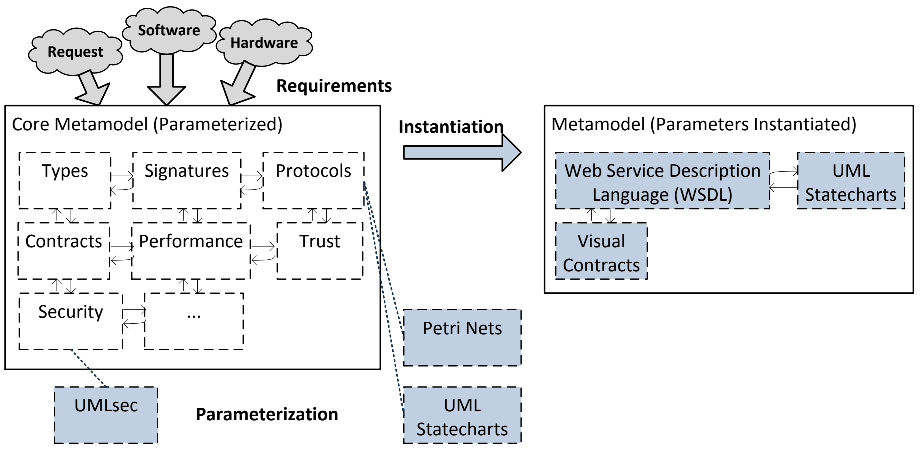


Fig. 1. Framework for the Integration of Modeling Languages

Core Language The core language contains abstract DSL-independent key concepts for each modeled service property. To determine them, requirements from the content of requests and service specifications have to be investigated. Based on DSLs used by domain experts, appropriate language constructs to represent these key concepts have to be identified. Depending on the separation of concerns, these language constructs have to be grouped into packages with explicit relations defined in the form of a metamodel algebra. These relations have to be checked for consistency throughout the core language that shall enable the correct integration of concepts from different part of the core metamodel.

Parameterization The parameterization is a possibility to substitute abstract parts of the core language with concrete syntaxes of DSLs. In Fig. 1, it is illustrated that either UML Statecharts [Obj10] or Petri Nets can be used for the specification of interaction protocols. The parameterization is enabled by defining a mapping between the core and a DSL that shall consider not only their syntax but also their semantics. So, a challenge of this work is to define the formal semantics of the core and develop a systematic approach to create semantical mappings from DSLs onto the core. This research will be based on the preliminary work on the formal definition of behavioral semantics by Soltenborn [SE] and a semantics-preserving mapping of DSLs by Semenyak [EKR$^+$].

Example The process of binding concrete DSLs onto the core is called instantiation. One instance of the core language that can be used for the comprehensive service specification is shown in the right box in Fig. 1. This is a language for rich service descriptions described by Huma in [HGEJ] that allows for service descriptions based on types, signatures, protocols, and contracts.

4 Conclusions and Further Steps

In this paper, a framework for the integration of modeling languages in the context of service-oriented computing is proposed. The framework represents key concepts of service aspects in the form of an abstract parameterized core language that enables to bind concrete DSLs onto it. The main advantages of the framework is the possibility to consistently use different DSLs in one comprehensive service specification and to match comprehensive service specifications written in different languages, both with the consideration of their syntax and semantics. This serves as a basis for the automatic discovery and composition of services in OTF Computing.

My work plan for the PhD is the following:

1. Develop the syntax of the core language that would fit the best for the binding of DSLs and matching of specifications.
2. Develop a formal semantics definition for the core language.
3. Identify packages of the core language and develop a metamodel algebra for relations between them. Check them for consistency.
4. Develop a systematic approach to map concrete DSLs on to the core syntactically and semantically. On the side of semantics, consider mathematical formalisms as well as some pragmatic approaches.
5. Evaluate the results.

The framework will be evaluated during the development of a service specification language in the context of the CRC 901 and by a student group who will build a framework for the composition of mobile applications.

References

[CCMW01] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Service Definition Language (WSDL). Technical report, March 2001.

[EKR$^+$] G. Engels, A. Kleppe, A. Rensink, M. Semenyak, C. Soltenborn, and H. Wehrheim. From UML Activities to TAAL - Towards Behaviour-Preserving Model Transformations. In *Proceedings of the 4th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2008)*, volume 5095 of *LNCS*, pages 95–109. Springer.

[HGEJ] Z. Huma, C. Gerth, G. Engels, and O. Juwig. A UML-based Rich Service Description for Automatic Service Discovery. In *Proceedings of the Forum at the CAiSE'12 Conference on Advanced Information Systems Engineering*, CEUR Workshop Proceedings, pages 90–97. CEUR-WS.org.

[Obj10] Object Management Group, Inc. (OMG). OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.3. Technical report, May 2010.

[SE] C. Soltenborn and G. Engels. Towards Test-Driven Semantics Specification. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS 2009)*, volume 5795 of *LNCS*, pages 378–392. Springer.

Automatized Generating of GUIs for Domain-Specific Languages

Michaela Bačíková, Dominik Lakatoš, and Milan Nosáľ

Technical University of Košice, Letná 9, 04200 Košice, Slovakia,
(michaela.bacikova, dominik.lakatos, milan.nosal)@tuke.sk,

Abstract. Domain-specific languages (DSLs) promise many advantages over general purpose languages (GPLs) and their usage is on the rise. That is one of the reasons for us at our university to teach the subject called Modelling and Generating of Software Architectures (MaGSA), where the students learn how to work with DSLs and their parsers. Our students often have problems understanding a new DSL, even if it is very simple, because they never have worked with a DSL before. Therefore we decided to find a way to help them with the learning process by providing a DSL-specific editor. This editor will be automatically generated based on a language specification and the generator will be integrated into the existing infrastructure of the MaGSA project, which the students use on the subject. In this paper we present various techniques to reach the goal of generating graphical editors for DSLs and we introduce a prototype which is able to automatically generate a user interface for an external DSL.

Keywords: domain-specific languages, graphical user interface, generative programming, metamodelling

1 Introduction

DSLs [1] promise advantages over GPLs. As mentioned by Brooks [2], programmers tend to produce statements at the same rate whatever language they use, so high-level languages are more productive. Still, only a few programmers go to the trouble of creating a new language as it comes with a plenty of problems lengthening the development process. It is hard to get it right at the first time without the proper knowledge: it is a new syntax to learn and maintenance can be expensive [3]. This usually leads to systems written in GPLs which, however, do not support domain-specific abstractions very well. A programmer ordinarily ends up with a code containing a lot of computer-specific programming noise and a lot less of the written code is really useful for solutions of domain problems.

At our university we teach a subject called Modelling and Generating of Software Architectures (MaGSA) which tries to show our students how to use model-based [4, 5] and generative programming [6, 7] for creating software systems. On this subjects' exercises the students use a pre-prepared project and tutorials, all constructed in a way enabling them to understand both the complexity of developing a rather simple language parser with several application generators and the simplicity of using a DSL to configure the desired application. It shows the possibilities of such an application generator tool to create different software applications by just changing the input file and running the implemented parsers and generators. We call the DSL used on the exercises the *entity language.* Metamodel of this DSL is defined by Java classes and its grammar is defined by an internal DSL (i.e. annotations) in the project and the *YAJCo (Yet Another Java Compiler Compiler)* parser generator generates a language processor for this DSL. YAJCo tool and it's advantages were completely described by Porubän [8, 9].

In our exercises we deal with problems related to incorrect understanding of the subjects' curriculum by our students. The primary problem is that the students do not get the picture of the syntax and the semantics of the entity language defined in our project i.e. they use it incorrectly and it takes some time for them to fully understand it and learn to work with it. Since no specific editor for the entity language was created yet, the students work with free text editors such as Notepad or Notepad++ which do not provide any syntax highlighting for our DSL and it has no code completion. So they do not give the students any hint for writing the code in the entity language. Therefore an idea arises to create a specific editor for the entity language, to be able to help the students learning a new language and to write code in it easily.

There is however still one important feature, namely the possibility of changing the DSL specification by changing the classes and annotations in the project - if it is because of changes in the subject curriculum or simply because of language evolution [10]. If the generator should be generally usable we can not give the students a one-off tool for a single DSL, which can not be changed. Moreover, the entity language specification really changes during the development process on the exercises. Therefore we decided to perform an experiment of generating a specific editor for the entity language. The main idea of the solution is similar to the idea of language workbenches described by Fowler [1, 11]. However the main difference in our approach is that the desired output is not a classical editor allowing the common integrated development environments like automatic code completion or syntax highlighting. It is rather a graphical interface with a support for creating new language constructions similar to CRUD (Create Retrieve Update Delete) interfaces.

There are many existing language workbenches that could help with graphical environment, code completion, syntax highlighting or refactoring features, for instance Eclipse Modeling Framework [14] and Eclipse Graphical Modeling Project [15], Spoofax [13], MetaEdit+ [16] or the Generic Modeling Environment [17]. The main reason for us to decide not to use them and go for the way of generating our own editor with a CRUD interface is *simplicity* and *understandability.* The students would learn to create new sentences in the language and they also would learn the relations that the user interface represents. The second reason is that these workbenches have integrated generators and language parsers, which are not compatible with the project used on our exercises and we want our students to create their own application generators which is the goal of the subject. The advantage is that once the editor generator is included in the MaGSA project, the students will have a chance to take a look into the techniques of generating simple editors for DSLs which they can further use in the subject. We present this paper as another solution to an existing problem of DSL learnability and we think that graphical user interfaces (GUIs) for DSLs represent a good first step for motivating students to use DSLs.

1.1 Tasks and goals

In this paper we try to introduce a first step on a way to the solution of automatic generating of GUIs for external DSLs. The possibility of derivating the user interface specific for a DSL based on its specification was already proved. We, however, try to present a different way of solving this problem which would shorten learning time of DSLs in the first phases of learning. Our general goal is to implement a prototype which will be able to generate graphical user interface based on a language specification. Based on different types of statements in the language specification it will be able to generate different basic graphical components. We will introduce various partial techniques to creating this solution and also different techniques to reach the general goal. The partial techniques will be illustrated by examples.

The goal of this paper is to design a method and to implement a prototype enabling automatic generating of a user interface for DSL based on language metamodel. The main goal is a definition of a solution, that will guide solvers to the right direction to the realization of a project. The most important property of our solution is simplicity and learnability.

2 The MaGSA Project

The prototype is based on an existing project, which is currently in the teaching process of the MaGSA subject on the Technical University of Koice, Faculty of Electrotechnical Engineering and Informatics for teaching metamodelling, DSLs and generative approach to software development. For the purpose of simplification, in the rest of the paper we will refer to this project as to the 'MaGSA project'. The MaGSA project used by our students is a pre-prepared project implemented in the Java language and it uses the standard MDA architecture. The exercises are realized in a form of a tutorial, which provide the description of the project and introductory source codes, with which the students work. We kindly ask the readers to check the official website in English [12] for further information about the MaGSA project and the tutorials.

Two basic entities are defined in the model of MaGSA project: Employee and Department. The model of the MaGSA project is written in the entity language and its metamodel can be described by a grammar as follows:

```
Model ::= ((Entity (Entity)*) (Reference)*)
Reference ::= (<reference> <NAME> <NAME>)
```

```
Entity ::= (<entity> <NAME> (<{> (Property (Property)*) <}>))
Property ::= (<NAME> (<:> Type) ((Constraint ((<,> Constraint))*))?)
Type ::= (<INTEGER> | <REAL> | <STRING>)

Constraint ::= (Regex | Required | Range | Length | Contains)
Regex ::= (<regex> <STRING\_VALUE>)
Required ::= <required>
Range ::= (<range> <INT_VALUE> <INT_VALUE>)
Length ::= (<length> <INT_VALUE> <INT_VALUE>)
Contains ::= (<contains> <STRING_VALUE>)
```

Terminals are noted in brackets ⟨ ⟩.

The basic entities of the metamodel and the relations between them are defined by the class implementation. The concrete syntax is defined by means of annotations in the individual classes. Based on these classes and annotations the YAJCo tool generates the grammar along with the language processor for the DSL.

2.1 Entity Language

As we mentioned earlier, the model of the generated application, is defined by an external DSL called the *entity language*, usage of which can be seen on an example file written in the entity language below.

```
# Department
entity Department {
    name : string required, length 5 30
    code : string required, length 1 4
    level : real
}

# Employee
entity Employee {
    name : string required, length 2 30
    surname : string required, length 2 30
    age : integer
}
reference Employee Department
```

On the MaGSA subject the students work with these two entities, but it is possible to define any number of entities and references between them and we encourage students to do so. As we mentioned earlier, they have problems grasping the basic idea, therefore only a very small amount of students do so without any help.

In the current state of the MaGSA project, the input file written in the entity language is processed by the generated language processor and based on the information processed the resulting application is generated, which provides a console interface to be able to work with the concrete departments and employees saved in the database. The task of the students is to create generators which generate the console application.

The students work with the entity language only manually which, when working with a big number of entities, could be tedious and exhausting. The problem can arise also when the metamodel is changed. In that case manual correction of each entity in the entity language file is needed. Therefore tool enabling working with the entity language would be helpful. The tool should enable adding new entities, their editing and removing. This tool would be generated based on the metamodels' grammar specifically for any defined DSL.

3 Problem Analysis

The challenge of our goal is the creation of a new generator of a DSL-specific GUI which would reflect the relations between the language concepts and thus support the domain-specific abstractions. The basic idea is best to explain on the grammar shown in the chapter 2.

The left sides of the rules define the points, which can be expressed by forms of the user interface. The right sides of the rules define the content of the forms.

In the EBNF we know these tree basic operators:

- sequence,
- alternative,
- repetition 0-* or 1-*,

and two basic types of symbols:

- terminal,
- non-terminal.

A sequence of symbols transforms into a GUI as a list of elements in a particular form. If the sequence contains terminal symbols, then these symbols have their type. Based on this type the corresponding component can be derived as illustrated in Tab. 1.

Table 1. Types of grammar symbols and components derived based on these types

Value type	*Derived component*
<STRING_VALUE>	text field
<INTEGER_VALUE>	jspinner or a number field (a text field with a number input)
<NAME>	selection with existing named entities

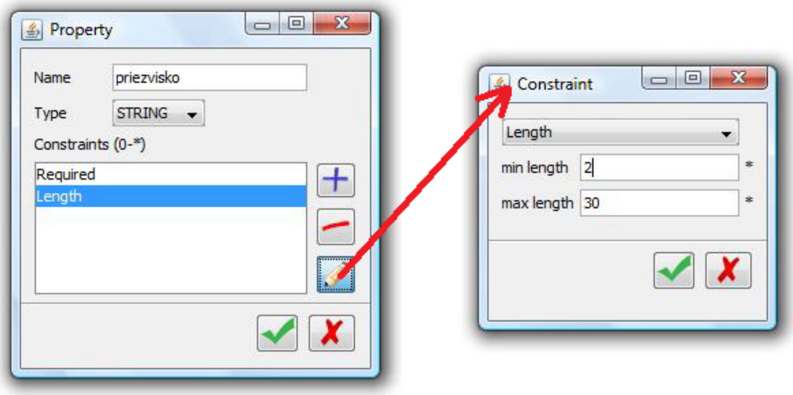
If a given symbol is a nonterminal then after clicking on a particular GUI element or an item a new dialog for editing this element or item opens.

A reduction of the number of dialogs can be made. If a rule contains only alternatives and no other elements or operators then it is possible to generate a dialog with a combo-box on the top and the remaining content will change dynamically based on what was selected in the combo-box. The content represents the particular alternative.

The following example shows two production rules for Property and Constraint.

```
Property ::= (<NAME> (<:> Type) ((Constraint ((<,> Constraint))*))?)
Constraint ::= (Regex | Required | Range | Length | Contains)
```

Fig. 1 illustrates a GUI for the `Constraint` and `Property` rules of our grammar. In the Property dialog new constraints can be added by clicking the '+' button. Consequently the dialog with one of the constraint types defined for the property can be chosen an its content automatically changes based on the particular alternative representation.



Property

Name: priezvisko

Type: STRING

Constraints (0-*)

Required

Length

Constraint

Length

min length: 2 *

max length: 30 *

Fig. 1. Illustration of a GUI generated from the Property grammar rule

Based on each type of EBNF operator it is possible to derive different types of components. Tab. 2 lists the components that can be derived based on a particular EBNF operator.

We list different types of components for each operator that can be used for the implementation, because they meet the same purpose. Each of them can be chosen for the implementation, the only deciding factor is usability.

As can be seen on these examples, the components reflect the relations between the language concepts. It can be seen instantly that the Property has a name, a type and a list of constraints. We assume this can help grasping the idea of DSLs.

Table 2. The types of operators in the grammar and components derived based on these types

Operator	*Derived component*
Alternatives (the — operator)	Combo-box
+ (repeat 1-*)	A combo-box or a radio button (with single selection) + 3x CRUD button
* (repeat 0-*)	A list or check-box buttons (with multiple selection)

4 Prototype Implementation

A prototype was implemented in the Java language into the existing infrastructure of the MaGSA project. A new generator was created by extending the abstract generator provided by the project. The metamodel is defined by the metamodel classes and annotations for the YAJCo parser generator and a user interface is generated to enable the model defined by the entity language. It is possible to add new entities, edit existing entities and to remove entities. Also, in each entity a further change of all properties defined by the model is possible. It is possible to add references from one entity to other. The initial screen of the application is in Fig. 2. It contains a list of all defined entities and a list of references. It is possible to create new files in the entity language with the *.el extension, or open existing files.

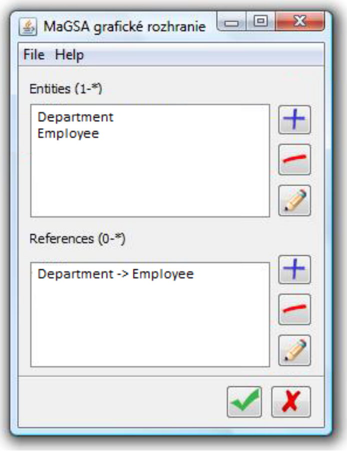


Fig. 2. Illustration of GUI generated from the grammar

We realize that this prototype is implemented in the specific environment of our MaGSA project, but the idea is usable also in other areas. We believe it will help beginners to learn the basic principles behind DSLs.

5 Conclusion

In this paper we introduced a first step on a way to the solution of automatic generating of GUIs for external DSLs. We showed a simple way of deriving a GUI for a DSL specification and of using the GUI to create new files written in this DSL. The DSL-specific interface is simple to use even for a student and it is more understandable than a classic textual editor. Reflecting relations between the domain-specific entities ensures the support of domain-specific abstractions. We introduced various partial techniques to creating this solution, illustrated by examples and also different techniques to reach the general goal. We created a prototype integrated into the existing infrastructure of the MaGSA project which is able to generate basic graphical elements based on the DSL specification. This way students will be guided to the right direction to the realization of the MaGSA project. In the future we plan to perform usability experiments with our students prototype to be able to improve the prototype further. The tool will be used by one group of students and another group will work without the tool. We will measure the learning time, completion time and failure rate. In the end we will evaluate subjective usability using questionnaires. We will compare the results between the two groups to determine whether the tool is really helpful.

6 Acknowledgement

This work was supported by VEGA Grant No. 1/0305/11 - Co-evolution of the artifacts written in DSLs driven by language evolution.

References

[1] Fowler, Martin: Domain-Specific Languages (Addison-Wesley Signature Series). Addison-Wesley Professional (2010)

[2] Brooks, F.P.: The Mythical Man Month: Essays on Software Engineering. Addison Wesley Professional Longman Aniversary Ed. (1995).

[3] Freeman, S., Pryce, N.: Evolving an Embedded Domain-Specific Language in Java. OOBSLA'06 October 22-26, Portland, Oregon, USA (2006).

[4] T. Stahl, M. Voelter: Model-Driven Software Development: Technology, Engineering, Management. Wiley (2006).

[5] J. Greenfield, K. Short, S. Cook, S. Kent: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley (2004).

[6] Czarnecki, K. Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley Professional (2000).

[7] Chen, J.: Generators. Presentation. USA, University of Colorado, march $http : //wiki.lassy.uni.lu/se2c - bib_download.php?id = 1117$ (accessible 27.5.2009) (2003).

[8] J. Porubän, M. Sabo, J. Kollár and M. Mernik: Abstract syntax driven language development: defining language semantics through aspects, in Proceedings of the International Workshop on Formalization of Modelling Languages, New York, NY, USA, (2010).

[9] J. Porubän, M. Forgáč, M. Sabo and M. Běhálek: Annotation Based Parser Generator. Computer Science and Information Systems, vol. 7, no. 2, pp. 291-307, (2010).

[10] Bell, P.: DSL Evolution. Article. December http://www.infoq.com/articles/dsl-evolution (accessible 20.4.2011) (2009).

[11] Fowler, M.: Language Workbenches: The Killer-App for Domain Specific Languages? http://www.martinfowler.com/articles/languageWorkbench.html, (accessible 27.5.2009)(2005).

[12] Modelling and Generating of Software Architectures. Student tutorial, English version. Technical University of Koice, Faculty of Electrical Engineering and Informatics, Department of Computers and Informatics http://hornad.fei.tuke.sk/ bacikova/MaGSA (2010)

[13] The Spoofax Language Workbench http://strategoxt.org/Spoofax (accessible 05.08.2012)(last update 2012)

[14] Eclipse Modeling Framework http://www.eclipse.org/modeling/emf/ (accessible 05.08.2012) (projets' last update 2009)

[15] Graphical Modeling Project http://www.eclipse.org/modeling/gmp/ (accessible 05.08.2012) (projets' last update 2010)

[16] Domain-specific Modeling with MetaEdit+ http://www.metacase.com/ (accessible 05.08.2012) (projets' last update 2012)

[17] Generic Modeling Environment http://www.isis.vanderbilt.edu/Projects/gme/ (accessible 05.08.2012) (projets' last update 2008)

A Model-driven Approach for the Analysis of Multimedia Documents

Joel dos Santos[1], Christiano Braga[2], and Débora Muchaluat-Saade[1]

[1] Laboratório MídiaCom
{joel,debora}@midiacom.uff.br
[2] Language-oriented software engineering research group
cbraga@ic.uff.br
Computer Science Department
Universidade Federal Fluminense

Abstract. This paper proposes a model-driven approach for the analysis of multimedia documents. Structural and behavioral properties of a multimedia document are verified thus guaranteeing its well-formedness and conformance before deployment. Multimedia documents are interpreted as object model instances of a multimedia document metamodel. Structural properties are verified using consistency reasoning over an ontology representation of the given object model together with OCL invariant validation (i.e., the application of OCL invariants to the given object model). Behavioral properties are verified through model checking on the transition system associated to the given multimedia document. Both metamodel and user-defined behavioral properties are verified.

1 Introduction

Multimedia documents describe applications as a set of components, which represent media objects, and relationships among them, which define temporal and real-time constraints over components. Declarative *authoring* languages may simplify the definition of multimedia documents since they emphasize the description of a document rather than the implementation of its presentation.

Large multimedia documents, with many components, organized within quite rich and complex structures, may be ill-formed and fall victim of conflicting relationships, leading to an application whose presentation is not what the author desires. An example of an ill-formed document includes cycles in composition nesting, where a composition is a group of components and relationships. Examples of undesired behavior (e.g. [7, 12, 13]) are the non-termination and/or unreachability of parts of a given document and the concurrent use of system resources, like an audio channel or a space at the screen. Usually, authors test their documents by executing them in an attempt to identify undesired behaviors, an approach which is not *complete* since not all possible behaviors may be explored on potentially *ill-formed* models.

Model-driven development (MDD) is a software development approach where models are the main artifacts of the development process. Models are represented as instances of metamodels, which describe the syntax of a modeling language, and may be used to derive different software artifacts, such as code in a programming language or even other models in different abstraction levels. This derivation process is called model transformation, which relates modeling languages through metamodels. Under that perspective, MDD is seen as a transformation between modeling languages applied to particular models, as shown in Figure 1,

$$m \in M \xrightarrow{parsing} \hat{m} \in \mathcal{M} \xrightarrow{\tau} \hat{n} \in \mathcal{N} \xrightarrow{pretty\text{-}printing} n \in N$$

Fig. 1. The application of a model transformation $\tau : \mathcal{M} \to \mathcal{N}$

where M represents the concrete and $\mathcal{M}$ the abstract syntax of a source modeling language, N represents the concrete and $\mathcal{N}$ the abstract syntax of a target modeling language and τ the transformation between $\mathcal{M}$ and $\mathcal{N}$. The operation *parsing* represents a mapping where a model m produces an instance $\hat{m}$ of $\mathcal{M}$ and *pretty-printing* represents the inverse mapping. The notation $m \in M$ denotes that model m is (syntactically) *well-formed* with respect to metamodel M.

The objective of MDD is to increase the abstraction level of the development process, so that authors may focus on modeling rather than implementation. In the context of multimedia documents, this is precisely the objective of NCL [9] and other multimedia authoring languages such as SMIL [15] and HTML5 [16]. Being *language-driven*, as shown in Figure 1, MDD fits very nicely with the development of multimedia applications in general and in particular with the validation of multimedia documents.

As previously mentioned, the verification of behavioral properties, like the ones presented before, is an important task in the development of multimedia applications, since it guarantees important properties of multimedia documents. Automaton based techniques fit well in this type of verification through model-checking [5] which essentially defines a decision procedure for temporal formulae. The application of model checking to multimedia documents thus requires a formalization of its behavior as a labeled transition system (LTS) and a proper encoding of the desired behavioral properties as temporal formulae.

A question that arises then is: *how to guarantee that the model being verified correctly represents the multimedia document?*

Figure 2 shows MDD with the so-called transformation contracts [2–4]. A transformation contract is a specification of a model transformation defined as a model that relates the metamodels of two modeling languages. Formally, a transformation contract $\mathcal{K}$ for a model transformation $\tau : \mathcal{M} \to \mathcal{N}$ is the disjoint union $\mathcal{M} \bowtie_{\mathcal{A}} \mathcal{N}$ of the metamodels of the modeling languages together with associations in $\mathcal{A}$ between the model elements of $\mathcal{M}$ and $\mathcal{N}$. The associations in $\mathcal{A}$ may be constrained by different kinds of properties, either structural or behavioral. The idea is that, following a *design by contract* style, every time a model transformation $\tau : \mathcal{M} \to \mathcal{N}$ is applied to a model $\hat{m} \in \mathcal{M}$, first the properties $P_{\mathcal{M}}$ of the source metamodel are verified in $\hat{m}$, then properties $P_{\mathcal{N}}$ of the target metamodel are verified in the generated model $\hat{n} = \tau(\hat{m})$, together with the properties $P_{\mathcal{K}}$ attached to the relations specified by $\mathcal{A}$, which must hold on the joined model $k = \hat{m} \bowtie_l \hat{n}$ with $l \in \mathcal{A}$. The notation $m \models P_{\mathcal{M}}$ means that the properties $P_{\mathcal{M}}$ hold in model m, that is, m *is in conformance* with $\mathcal{M}$.

$$m \in M \xrightarrow{parsing} \begin{array}{c}\hat{m} \in \mathcal{M},\\ \hat{m} \models P_{\mathcal{M}}\end{array} \xrightarrow{\tau} \begin{array}{c}\hat{n} \in \mathcal{N},\\ \hat{n} \models P_{\mathcal{N}},\\ k \models P_{\mathcal{K}}\end{array} \xrightarrow{pretty\text{-}printing} n \in N$$

Fig. 2. MDD with transformation contracts

An MDD process for the analysis of multimedia documents would refine the process of Figure 2 by understanding $\mathcal{M}$ as the metamodel of a multimedia authoring language, such as NCL, and $\mathcal{N}$ as the metamodel of the specification language of a formal verification framework such as the specification language of a model checker.

This work proposes a transformation contract approach for the analysis of multimedia documents. Different verification techniques shall be used to analyze multimedia documents: (i) Consistency reasoning with description logic [1] will be used for verifying document consistency together with Object Constraint Language (OCL) invariant execution; and (ii) Linear Temporal Logic model checking appears to be the appropriate reasoning technique for behavioral properties of multimedia documents.

This work contributes with a general framework, with tool support, capable of analyzing different types of multimedia documents using different analysis (that is, verification and validation) techniques. Our proposal uses a language-driven approach where the authoring language semantics is represented by a general model (called SHM - Simple Hypermedia Model) where structural and behavioral properties are verified. In this paper we outline our approach and discuss preliminary results achieved with a prototype of the tool.

The remainder of this paper is organized as follows. Section 2 presents the state-of-the-art on multimedia document analysis. Section 3 discusses the proposed solution for multimedia document analysis. Section 4 discusses the current state of the multimedia document analysis project illustrating preliminary results. Section 5 finishes this paper presenting the next steps of this work.

2 State-of-the-art on multimedia document analysis

Santos et al. [13] presented an approach for the analysis of multimedia documents by translating it into a formal specification, in that case, into RT-LOTOS processes, using general mapping rules. The modularity and hierarchy of RT-LOTOS allows the combination of processes specifying the document presentation with other processes modeling the available platform.

The verification consists in the interpretation of the minimum reachability graph built from the formal specification, in order to prove if the action corresponding to the presentation end can be reached from the initial state. Each node in the graph represents a reachable state and each edge, the occurrence of an action or temporal progression. When a possible undesired behavior is found, the tool returns an error message to the author, so he can repair it. The tool in [13] could analyze NCM [14] and SMIL [15] documents.

Na and Furuta, in [12], presented caT (context aware Trellis), an authoring tool based on Petri nets. caT supports the analysis of multimedia documents by building the reachability tree of the analyzed document. The author defines limit values for the occurrence of dead links (transitions that may not be triggered), places with token excess, besides other options, as the analysis maximum time. The tool investigates the existence of a terminal state, i.e., whether there is a state where no transitions are triggered. It also investigates the limitation property, i.e., if no place in the net has an unlimited number of tokens and the safeness property, i.e., if each place in the net has a token. The limitation analysis is important since tokens may represent scarce system resources.

Oliveira et al., in [7], presented HMBS (Hypermedia Model Based on Statecharts). An HMBS multimedia application is described by a statechart that represents its structural hierarchy, regarding nodes and links, and its *human-consumable* components. Those components are expressed as information units, called pages and anchors. The statechart execution semantics provide the application navigation model. A statechart state is mapped into pages and transactions and events represent a set of possible link activations.

The statechart reachability tree for a specific configuration may be used to verify if any page is unreachable, by verifying the occurrence of a state s in one of the generated configurations, which indicate that the page is visible when the application navigation starts in the initial state considered. In a similar manner, it is possible to determine if a certain group of pages may be seen simultaneously searching state configurations containing the states associated to those pages. The reachability tree also allows the detection of configurations from which no other page may be reached or that present cyclical paths.

Júnior et al., in [10], also present the verification of NCL documents through a model-driven approach. The verification is also achieved by transforming an NCL document into a Petri Net. This transformation is done in two steps. The first step transforms the NCL document into a language called FIACRE, representing the document as a set of components and processes. Components represent media objects and compositions and processes represent the behavior associated to components. The second step transforms the FIACRE representation into a Petri Net. The verification uses a model-checking tool and temporal logic formulae to represent the behavior the author wants to verify. Once this work is very recent, the automation of that approach is a future work.

Our work contributes to the state-of-the-art with a *general* approach that can be used with different multimedia authoring languages.

3 A model-driven approach to multimedia document analysis

We propose the use of the transformation contracts approach to analyze multimedia documents. Figure 3 refines Figure 2 and illustrates our approach pictorially with NCL as the multimedia authoring language and Maude [6] as the specification language for formalizing multimedia documents. Informally, Maude modules are produced from NCL documents and the behavioral properties are represented as LTL formulae which are verified using the Maude model checker.

An important element of our approach is the so-called modeling language for the Simple Hypermedia Model (SHM) [8]. SHM models are important for two reasons: (i) they give formal meaning to NCM models, and (ii) should be a *general* formal representation for multimedia documents. SHM models are essentially transition systems that have basic elements to represent multimedia documents such as *anchors* as states, *events* as actions and *links* as transitions.From SHM models we could produce representations in different formalisms such as Maude or SMV [11]. Behavioral properties of well-formed models that hold the structural properties of a given authoring language are then checked at the concrete level such as Maude or SMV.

Let us go through each step of Figure 3. First, an NCL document is *parsed* into an NCM [14] model. (NCM is the conceptual model that NCL documents are based on and may be understood as its abstract syntax.) Thus, given an NCL document d, if $(\hat{d} = parse(d)) \models P_{\mathcal{NCM}}$, that is, if the structural properties of NCM hold in $\hat{d}$ (such as non-circular nested compositions) then a model transformation $\tau_{\mathcal{NCM}}$ is applied on $\hat{d}$. Given that a proper SHM model $\hat{s}$ is produced by the application of the transformation contract from NCM to SHM, that is, essentially, its states are built properly from anchors, actions properly built from events and transitions properly built from links, a concrete representation of $\hat{s}$ may be produced in the specification language of the model checker, such as Maude.

$$d \in NCL \xrightarrow{parsing} \begin{array}{l} \hat{d} \in \mathcal{NCM}, \\ \hat{d} \models P_{\mathcal{NCM}} \end{array} \xrightarrow{\tau_{\mathcal{NCM}}} \begin{array}{l} \hat{s} \in \mathcal{SHM}, \\ \hat{s} \models P_{\mathcal{SHM}}, \\ k \models P_{\mathcal{K}} \end{array} \xrightarrow{pretty\text{-}printing} m_d \in Maude$$

Fig. 3. A transformation contract approach to Maude theories from NCL documents

Given m_d, which is well-formed and in conformance with $K = \mathcal{NCM} \bowtie_{\mathcal{A}} \mathcal{SHM}$, one can now verify with a model-checker the temporal formulae that represent the behavioral properties exemplified at the beginning of Section 1 (such as unreachability of document parts) and document specific properties, defined by the document author and transformed into temporal formulae. Counterexamples produced by the model-checker, which are essentially traces that do not have the desired temporal formulae, may be presented back to the document author as sequences of links representing SHM transitions that correspond to transitions (or rewrites, in the case of Maude) of the faulty path encountered by the model checker. This process is illustrated pictorially in Figure 4, where

$$\begin{tikzcd} \mathit{NCL\ author} \arrow[r, shift left, "d \in NCL"] & \mathit{NCL\ Analyzer} = \tau(\mathit{parse}(d)) \vdash \mathit{modelCheck}(s_0, \phi) \arrow[l, shift left, "l \in (NCL_{Links})^*"] \end{tikzcd}$$

Fig. 4. NCL Analyzer

NCL Analyzer is the tool that essentially invokes the Maude model checker, represented in Figure 4 by the command *modelCheck*, which checks for the property ϕ (a conjunction of the behavioral properties together with author-defined properties) using the specification (actually, rewrite theory) given by $\tau(\mathit{parse}(d))$ using s_0 as initial state (specified by the initial conditions of document d).

As mentioned before, SHM is intended to be a general multimedia model. The verification of multimedia documents specified with languages different from NCL, such as SMIL and HTML5, would require transformations from the abstract syntax of those languages to *SHM* together with a proper mapping from counter-examples of the chosen model-checker to the authoring language. The remaining of the analysis process is reused among those different languages.

We have a first attempt at SHM and a prototype tool that transforms NCL to Maude modules. Section 4 briefly discusses preliminary results.

4 Preliminary results

Part of the proposed solution is prototyped in a tool presented in [8], where the first author, under the supervision of the remaining authors, proposed an implementation of a transformer from NCL documents to Maude modules. With that prototype it was possible to analyze structural and behavioral properties of NCL documents. Besides, the prototype gives us the intuition that the proposed solution seems to be appropriate.

The prototype was used in several small experiments with simple documents. Besides it was used with two non-trivial documents created by the Brazilian Digital TV community. A description of the two documents ("First João" and "Live More") and their results are presented here.

"First João" is an interactive TV application that presents an animation inspired in a chronicle about a famous Brazilian soccer player named Garrincha. It plays an animation, an audio and a background image. At the moment Garrincha dribbles the opponent, a video of kids performing the same dribble is presented and when his opponent falls on the ground, a photo of a kid in the same position is presented. The user may interact with the application pressing the red key at the moment a soccer shoes icon appears. The animation is resized and a video of a kid thinking about shoes starts playing.

This document was deployed by the authors of the NCL language as a sample document. As expected, the document is consistent with respect to the structural properties ($P_{\mathcal{NCM}}$), defined taking into account the NCM grammar, and the behavioral properties ($P_{\mathcal{SHM}}$), from the set of parameterized properties. It was possible to verify that every anchor is reachable and has an end. Besides, the document as a whole ends.

"Live More" is an application that presents a TV show discussing health and welfare. Once the TV show starts playing, an interaction icon appears. If the user presses the red key of the remote control, four different food options appear. The user can choose a dish by pressing one of the colored keys of the remote control. When a dish is chosen, the TV user is informed about the quality of his choice, telling whether there are missing nutrients or nutrients in excess.

This document is consistent with respect to the structural properties ($P_{\mathcal{NCM}}$). However, the document is not consistent with respect to the behavioral properties ($P_{\mathcal{SHM}}$). It was possible to verify that once a dish is chosen, the anchor representing the chosen dish and its result do not end, and consequently the document as a whole.

The proposed prototype allows NCL document authors to verify if their document fails in one of the common undesired properties, besides validating the document structure. From the tests done with NCL documents it was possible to identify refinements in our Maude specification of $\mathcal{SHM}$. Such refinements and open issues are addressed in the next section.

5 Conclusion

In this paper we presented an approach for the analysis of multimedia documents and a prototype tool that partially implements it. This section discusses future directions to our research project.

We are currently working on a refinement of the specification for $\mathcal{SHM}$ in [8], its Maude representation (to improve the efficiency of model checking it) and on a formal proof for the transformation $\tau_{\mathcal{NCM}}$.

An important future work it to evaluate the generality of our approach, exploring mappings from different authoring languages to $\mathcal{SHM}$, as indicated in the end of Section 3.

Our preliminary results consider predefined properties representing patterns of behavior of multimedia documents (see Section 3.) We plan to incorporate user-defined behavioral properties, by allowing the author to define such properties in a *structured* natural language (English, for example) that could be translated to LTL formulae.

We also consider evaluating the usability of the tool resulting from this project using human-computer interaction techniques.

References

1. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook.* Cambridge University Press, 2003.

2. C. Braga. A transformation contract to generate aspects from access control policies. *Journal of Software and Systems Modeling*, 10(3):395–409, 2010.
3. C. Braga, R. Menezes, T. Comicio, C. Santos, and E. Landim. On the specification verification and implementation of model transformations with transformation contracts. In *14th Brazilian Symposium on Formal Methods*, volume 7021, pages 108–123, 2011.
4. C. Braga, R. Menezes, T. Comicio, C. Santos, and E. Landim. Transformation contracts in practice. *IET Software*, 6(1):16–32, 2012.
5. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.
6. M. Clavel, S. Eker, F. Durán, P. Lincoln, N. Martí-Oliet, and J. Meseguer. *All about Maude - A High-performance Logical Framework: how to Specify, Program, and Verify Systems in Rewriting Logic*. Springer-Verlag, 2007.
7. M.C.F. de Oliveira, M.A.S. Turine, and P.C. Masiero. A statechart-based model for hypermedia applications. *ACM Transactions on Information Systems*, 19(1):52, 2001.
8. J. A. F. dos Santos. Multimedia and hypermedia document validation and verification using a model-driven approach. Master's thesis, Universidade Federal Fluminense, 2012.
9. ITU. Nested Context Language (NCL) and Ginga-NCL for IPTV services. http://www.itu.int/rec/T-REC-H.761-200904-S, 2009.
10. D. P. Júnior, J. Farines, and C. A. S. Santos. Uma abordagem MDE para Modelagem e Verificação de Documentos Multimídia Interativos. In *WebMedia*, 2011. in Portuguese.
11. K.L. McMillan. *Symbolic model checking: an approach to the state explosion problem*. Kluwer Academic Publishers, 1993.
12. J.C. Na and R. Furuta. Dynamic documents: authoring, browsing, and analysis using a high-level petri net-based hypermedia system. In *ACM Symposium on Document engineering*, pages 38–47. ACM, 2001.
13. C.A.S. Santos, L.F.G. Soares, G.L. de Souza, and J.P. Courtiat. Design methodology and formal validation of hypermedia documents. In *ACM International Conference on Multimedia*, pages 39–48. ACM, 1998.
14. L. F. G. Soares, R. F. Rodrigues, and D. C. Muchaluat-Saade. Modeling, authoring and formatting hypermedia documents in the HyperProp system. *Multimedia Systems*, 2000.
15. W3C. Synchronized Multimedia Integration Language - SMIL 3.0 Specification. http://www.w3c.org/TR/SMIL3, 2008.
16. W3C. HTML5: A vocabulary and associated APIs for HTML and XHTML. http://www.w3.org/TR/html5/, 2011.

SMADL: The Social Machines Architecture Description Language

Leandro Marques do Nascimento[1,2], Vinicius Cardoso Garcia[1], Silvio R. L. Meira[1]

[1] Informatics Center - Federal University of Pernambuco (UFPE) ,
[2] Department of Informatics - Federal Rural University of Pernambuco (UFRPE)
{lmn2, vcg, srml}@cin.ufpe.br

Abstract. We are experiencing a high growth in the number of web applications being developed. This is happening mainly because the web is going into a new phase, called programmable web, where several web-based systems make their APIs publicly available. In order to deal with the complexity of this emerging web, we define a notion of social machine and envisage a language that can describe networks of such. To start with, social machines are defined as tuples of input, output, processes, constraints, states, requests and responses; apart from defining the machines themselves, the language defines a set of connectors and conditionals that can be used to describe the interactions between any number of machines in a multitude of ways, as a means to represent real machines interacting in the real web. This work presents a preliminary version of the Social Machine Architecture Description Language (SMADL).

1 Introduction

Software systems are built upon programming languages. A programming language is a notation for expressing computations (algorithms) in both machine and human readable form. Appropriate languages and tools may drastically reduce the cost of building new applications as well as maintaining existing ones [1].

In the context of programming languages, a Domain-Specific Language (DSL) is a language that provides constructs and notations tailored toward a particular application domain [2]. Usually, DSLs are small, more declarative than imperative, and more attractive than General-Purpose Languages (GPL) for their particular application domain.

However, in software engineering several different artifacts are developed besides code and one of the most important is the software architecture. Most developers agree that architecture is needed in some way, shape, or form, but, they can't agree on a definition, don't know how to manage it efficiently in nontrivial projects, and usually can't express a system's architectural abstractions precisely and concisely [3]. When asking a developer to describe a system's architecture Voelter [3] says "*I get responses that include specific technologies, buzzwords* *such as AJAX (asynchronous JavaScript and XML) or SOA (service-oriented architecture), or vague notions of "components" (such as publishing, catalog, or payment). Some have wallpaper-sized UML diagrams in which the meanings of the boxes and lines aren't clear.*"

These answers mention aspects that are actually related to a system's architecture, but none of them represent an unambiguous and/or "formal" description of a system's core abstractions. Indeed, it is not surprising because, although there are languages that directly express software architectures, they are not quite common among software developers.

In order to better define software architectures, it is worthy using DSLs and taking advantage of their expressiveness in a limited domain. Our proposal relies on top of an Architecture Description Language (ADL) for describing web based software systems in terms of Social Machines, a new concept developed by our research group which tries to increase the abstraction level for comprehending the web. Next, we present the context and the details about Social Machines.

2 An Emerging Web of Social Machines

The traditional concept of software has been changing during the last decades. Since the first definition of a computing machine described by Turing in [4], software started to become part of our lives and has been turned pervasive and ubiquitous with the introduction of personal computers, the internet, smartphones and recently the internet of things. In fact, one can say that software and the internet changed the way we communicate, the way business is done and the way software is developed, deployed and used. Nowadays, computing means connecting [5] and sometimes it is said that developing software is the same as connecting services [6], since there are several up and running software services available.

Recently, we all can clearly see that a new phase is emerging, the web "3.0", the web as a programming platform, the network as an infrastructure for innovation, on top of which all and sundry can start developing, deploying and providing information services using the computing, communication and control infrastructures in a way fairly similar to utilities such as electricity.

An overview of this Web 3.0 scenario can be seen in the *ProgrammableWeb* website[1]. It gathers around 6500 publicly available APIs and more than 6700 mashups using them (last visit in July 2012). Although there have been many studies about the future of the internet and concepts such as web 3.0, programmable web [7, 8], linked data [9] and semantic web [10, 11], the segmentation of data and the issues regarding the communication among systems obfuscates the interpretation of this future. Unstructured data, unreliable parts and non-scalable protocols are all native characteristics of the internet that needs a unifying view and explanations in order to be developed, deployed and used in a more efficient and effective way.

[1] www.programmableweb.com

Furthermore, the Web concepts, as we know, are recent enough to represent many serious difficulties while understanding their basic elements and how they can be efficiently combined to develop real, practical systems in either personal, social or enterprise contexts. Therefore, we developed a new concept called Social Machine (SM), in order to provide a common and coherent conceptual basis for understanding this still immature, upcoming and possibly highly innovative phase of software development. SM concept was firstly conceived in [12] and later demonstrated with a case study in [13].

So, we define a SM as a tuple, as following:

```
SM = <Rel, WI, Req, Resp, S, Const, I, P, O>
```

In general, a SM represents a connectable and programmable entity containing an internal *processing unit* **(P)** and a *wrapper interface* **(WI)** that waits for *requests* **(Req)** from and replies [with *responses* **(Resp)**] to other social machines. Its processing unit receives *inputs* **(I)**, produces *outputs* **(O)** and has *states* **(S)**; and its connections define intermittent or permanent *relationships* **(Rel)** with other SMs. These relationships are connections established under specific sets of *constraints* **(Const)**. Our goal with this concept of a Social Machine is not to formally describe software services as can be seen in [14], but instead we want to describe the programmable web in a higher level of abstraction, thus increasing the power of new programming structures or paradigms dedicated to this context. Figure 1 illustrates a basic representation of a Social Machine.

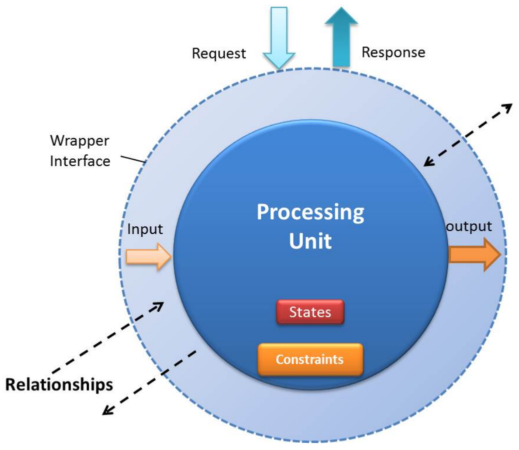


Fig. 1. A graphical representation of a Social Machine.

The idea behind Social Machines is to take advantage of the networked environment they are in to make it easier to combine and reuse exiting services from different SMs and use them to implement new ones. Hence, we can highlight some of its main characteristics, as following: *Sociability, Compositionality,* *Platform and Implementation independency, Self-awareness, Discoverability* and last, but not least, *Programmability.*

There may be different types of social machines, but one way to classify them is through the simple taxonomy shown in Figure 2, based on the types of interactions they have with each other, as follows:

– **Isolated** - Social Machines that have no interaction with other Social Machines;
– **Provider** - Social Machines that provide services for other Social Machines to consume;
– **Consumer** - Social Machines that consume services that other Social Machines provide;
– **Prosumer** - Social Machines that both provide and consume services.

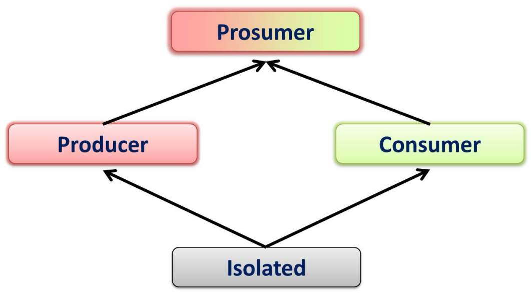


Fig. 2. Social Machines as a partial order diagram.

In this work, we envisage an Architecture Description Language that can describe networks of SMs. Apart from defining the machines themselves, the ADL defines a set of connectors and conditionals that can be used to describe the interactions between any number of machines in a multitude of ways, as a means to represent real machines interacting in the real web. Details are presented next.

3 The Social Machines Architecture Description Language - SMADL

This work is an attempt to answer the following research questions: "*Is it possible to integrate diverse web applications using a standard architecture description language?*". In order to answer it, this work purposes a new ADL for defining social machines: SMADL.

Social Machines can be connected (or establish a relationship) in basically two phases: in the first phase, the SMs must find each other, and a there must be a SM registry service much likely Internet DNS; in the second phase the SMs actually connect to each other and exchange information for a limited period of time. The SM registry service is out of the scope of this proposal. We are assuming SMs can find each other without much effort.

In order to comprise these two phases, SMADL is composed by two mini-languages:

– **VCL** (*Visitor Card Language*): presents the externally visible properties of a SM, i.e., which requests and responses it accepts, which types of inputs/outputs it handles, if an internal state is maintained and/or how many requests it can handle per amount of time. A vCard is provided by the SM registry to the consumer SM, so it can decide if the relationship is interesting or not. Business issues are also present in a SM vCard, such as, billing information and service level agreements. Nowadays, popular web APIs do not make available such business information in a programmatic way. Usually, there are only few lines in reduced font size contracts mentioning that important information.

– **WIL** (*Wrapper Interface Language*): we are assuming every SM has a vCard with which the wrapper interface is fully complaint. This language is responsible for actually connecting SMs, establishing pre and post conditions, applying different connectors in a SM composition, and implementing business rules associated with a given set of SMs. Our proposal is to use WIL not as substitute for the currently available technologies. Instead it increases the level of abstraction of these technologies, freeing the programmer to concentrate on business issues of the SM relationships.

To understand better how these mini-languages are used, Figure 3 shows the steps for establishing a relationship between two SMs, as following:

1. Initially, the requester SM, in this case represented by Evernote (upper left icon), searches for some SM registered as a micro blog, in our case Twitter (upper right icon). Note that Evernote presents its vCard to DNS while searching for a service, once the responder may or may not accept connections with that specific SM.
2. The SM DNS finds Twitter and requests its vCard.
3. Twitter responds with its vCard, accepting the relationship.
4. The SM DNS replies back to Evernote with the Twitter vCard, which includes its address.
5. Using Twitter vCard and knowing its wrapper interface, Evernote establishes a relationship with Twitter, following all conditions imposed.

There are several popular technologies for integrating web based or service oriented systems, such, REST [15] and OSGi [16]. The current version of SMADL generates code for REST based apps, as it is becoming the most popular on the web, adopted by big players such as Facebook and Google. According to ProgrammableWeb site 4300 out of approximately 6500 APIs uses REST as base technology.

SMADL is being developed on Xtext language workbench [17]. As this is a work in progress, we are preliminarily evaluating alpha versions of the language and planning an experiment using the approach proposed by [18].

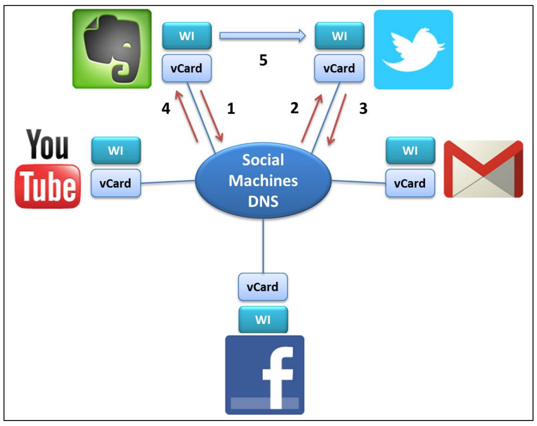


Fig. 3. Steps for establishing a relationship between two example Social Machines.

4 Related Work

We performed a systematic mapping study [19] for better understanding the DSL/ADL research field as shown in [20]. Initially, 4450 studies were identified, and, after filtering, 1440 primary studies were selected and categorized. Among all those primary studies, different methods/techniques for handling DSLs (creating, evolving, maintaining, testing) could be listed and several DSLs applied to several different domains could be identified. The domain where DSLs are most frequently applied is the Web domain. Other domains such as embedded systems, data intensive apps, and control systems where quite common too.

In our study we could enumerate 30 publications directly related to ADL. Amongst them, only two of them mention the Web domain, both from 2010. In the first one [21], the authors propose to formalize the architectural model using domain-specific language, an ADL which supports the description of dynamic, adaptive and evolvable architectures, such as SOA itself. Their ADL allows the definition of executable ver-sions of the architecture. The second one is [**?**] which presents a framework for the implementation of best practices concerning the design of the software architecture. The authors present an implementation of the framework in the Eclipse platform and an ADL dedicated to Web applications.

In addition, practical examples, such as *Yahoo! Pipes*[2] and *IfThisThenThat*[3] can be seen as related work. The former uses a graphical tool for customizing data flows from different sources. The latter allows end users to program the web based on pre-defined events fired by a set of channels, for example, if someone tags you on a given social network (channel 1), then save this photo in the person's virtual drive (channel 2). The user can choose among different events from different channels, which are, in practice, websites that make available their APIs. Our work is an attempt to be a completely different way to program

[2] http://pipes.yahoo.com
[3] http://ifttt.com/

the Web, not based on pre-defined parameters. The idea behind SMADL is to actually define every public API in the Web and the relationships among them. This way, composition possibilities for several SMs can be infinite.

As can be seen, this is a relatively new research field and we believe we can make a considerable contribution by establishing the concept of a Social Machine and developing an ADL for supporting it.

5 Concluding Remarks and Future Work

This work presents SMADL "The Social Machines Architecture Description Language" as a possible solution for modeling web-based software systems.

In general, a Social Machine (SM) represents a connectable and programmable entity containing an internal *processing unit* **(P)** and a *wrapper interface* **(WI)** that waits for *requests* **(Req)** from and replies [with *responses* **(Resp)**] to other social machines. Its processing unit receives *inputs* **(I)**, produces *outputs* **(O)** and has *states* **(S)**; and its connections define intermittent or permanent *relationships* **(Rel)** with other SMs. These relationships are connections established under specific sets of *constraints* **(Const)**.

Our main goal is to use SMADL to describe SM relationships and then we can have one unique way to program the web, independently of what technology/platform is being used. The current version (alpha) of SMADL generates code for REST based apps, as it is becoming the most popular on the web, adopted by big players such as Facebook and Google. Nowadays, we are working on several sample apps which have their architecture written in SMADL. These apps are basically consumer SMs, it means they do not make available public features, but only consumes other public APIs. These apps are going to be distributed as open source.

Our next steps include writing fully prosumer social machines, i.e. applications that connects to others, process their data, and someway make this data available for others to consume. At this phase, we are planning to perform an experiment, following the methodology described in [18].

Acknowledgments. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES), funded by CNPq and FACEPE, grants 573964/2008-4, APQ-1037-1.03/08 and APQ-1044-1.03/10 and Brazilian Agency (CNPq processes number 475743/2007-5 and 140060/2008-1).

References

1. Pressman, R.S.: Software engineering: a practitioner's approach (2nd ed.). McGraw-Hill, Inc., New York, NY, USA (1986)
2. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. **37**(4) (December 2005) 316–344

3. Völter, M.: Architecture as language. IEEE Software **27**(2) (2010) 56–64
4. Turing, A.M.: On computable numbers, with an application to the entscheidungsproblem. Proceedings of the London Mathematical Society **42** (1936) 230–265
5. Roush, W.: Social Machines. Technology Review (2006) 1–18
6. Turner, M., Budgen, D., Brereton, P.: Turning software into a service. Computer **36**(10) (October 2003) 38–44
7. Yu, S., Woodard, C.J.: Service-oriented computing — icsoc 2008 workshops. Springer-Verlag, Berlin, Heidelberg (2009) 136–147
8. Hwang, J., Altmann, J., Kim, K.: The structural evolution of the web 2.0 service network. Online Information Review **33**(6) (2009) 1040–1057
9. Bizer, C., Heath, T., Berners-Lee, T.: Linked data - the story so far. Int. J. Semantic Web Inf. Syst. **5**(3) (2009) 1–22
10. Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. Scientific American **284**(5) (2001) 34–43
11. Hitzler, P., Krtzsch, M., Rudolph, S.: Foundations of Semantic Web Technologies. 1st edn. Chapman & Hall/CRC (2009)
12. Meira, S.R.L., Burégio, V.A., Nascimento, L.M., de Figueiredo, E.G.M., Neto, M., Encarnação, B.P., Garcia, V.C.: The Emerging Web of Social Machines. CoRR **abs/1010.3** (2010)
13. Meira, S.R.L., Buregio, V.A.A., Nascimento, L.M., Figueiredo, E., Neto, M., Encarnacao, B., Garcia, V.C.: The Emerging Web of Social Machines. In: 2011 IEEE 35th Annual Computer Software and Applications Conference, IEEE (July 2011) 26–27
14. Broy, M., Krüger, I.H., Meisinger, M.: A formal model of services. ACM Trans. Softw. Eng. Methodol. **16**(1) (February 2007)
15. Richardson, L., Ruby, S.: Restful web services. First edn. O'Reilly (2007)
16. Hall, R.S., Pauls, K., McCulloch, S., Savage, D.: OSGi in Action: Creating Modular Applications in Java. Volume 188. Manning (2010)
17. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion. SPLASH '10, New York, NY, USA, ACM (2010) 307–309
18. Juristo, N., Moreno, A.: Basics of Software Engineering Experimentation. Springer (2001)
19. Petersen, K., Feldt, R., Mujtaba, S., Mattsson, M.: Systematic mapping studies in software engineering. In: Proceedings of the 12th international conference on Evaluation and Assessment in Software Engineering. EASE'08, Swinton, UK, UK, British Computer Society (2008) 68–77
20. Nascimento, L.M., Viana, D.L., da Mota Silveira Neto, P.A., Souto, S.F., Martins, D.A.O., Garcia, V.C., Meira, S.R.L.M.: Domain-Specific Languages - A Systematic Mapping Study. In: Proceeedings of 7th International Conference on Software Engineering Advances (ICSEA). (2012)
21. López-Sanz, M., Cuesta, C.E., Marcos, E.: Formalizing high-level service-oriented architectural models using a dynamic adl. In: Proceedings of the 2010 international conference on On the move to meaningful internet systems. OTM'10, Berlin, Heidelberg, Springer-Verlag (2010) 57–66

Verifiable composition of language extensions

Ted Kaminski

Department of Computer Science and Engineering
University of Minnesota, Minneapolis, MN, USA
`tedinski@cs.umn.edu`

Abstract. Domain-specific languages offer a variety of advantages, but their implementation techniques have disadvantages that sometimes prevent their use in practice. *Language extension* offers a potential solution to some of these problems, but remains essentially unused in practice. It is our contention that the main obstacle to adoption is the lack of any assurance that the compiler composed of multiple independent language extensions will work without the need for additional modifications, or at all. We propose to solve this problem by requiring extensions to independently pass a composition test that will ensure that any such extensions can be safely composed without "glue code," and we propose to demonstrate that interesting extensions are still possible that satisfy such a test.

1 Introduction

Domain-specific languages (DSLs) come with a variety of reasonably well-known advantages and disadvantages [3]. Some of these disadvantages do not seem to be inherent to DSLs in general, but are a consequence of the way they are implemented. In particular, many implementation techniques lack or poorly support *composition*, meaning multiple DSLs cannot easily be used together to solve a problem.

To be more precise about what we mean by language composition, we will use some of the classification and notation of Erdweg, Giarrusso, and Rendel [5]. The notation $H \triangleleft E$ represents a *host language* H composed with a *language extension* E, specifically crafted for H. Another composition operator $L_1 \uplus_g L_2$ denotes the composition of two distinct languages with "glue code" g. To permit only the $\triangleleft$ form of language composition ("language extension") is not sufficient. With H, $H \triangleleft E_1$, and $H \triangleleft E_2$, we are left with no option for composing all three, without modifying one of the extensions to have the form $(H \triangleleft E_1) \triangleleft E_2$ (or vice versa.) However, the $\uplus_g$ form of language composition ("language unification") is also insufficient for our purposes. The problem with this form of composition is that the "glue code" g necessary to perform this composition is essentially an admission that the composition is broken and must be repaired. (Though it is still interesting that the composition *can* be repaired.)

What we seek is a composition method $L_1 \uplus_\emptyset L_2$, that is, language unification without needing any glue code ($g = \emptyset$.) This may seem impossible in general, but there is hope in special cases, such as when both languages are extensions to a common host: $(H \triangleleft E_1) \uplus_\emptyset (H \triangleleft E_2)$. Here we are tasked with resolving only conflicts between E_1 and E_2, while the host language H is shared. We will say that a DSL implementation technique supports *composable language extension* if it is capable of composition of the form $H \triangleleft (E_1 \uplus_\emptyset E_2)$. We further require that the technique provides some assurance that the resulting composed language will work as intended, and is not simply broken.

The goal of this work is to build a DSL implementation tool and demonstrate that it satisfies the following criteria:

– Supports composable language extension, as defined above.
– Permits introduction of new syntax.
– Permits introduction of new static analysis on existing syntax.
– Capable of generating good, domain-specific error messages.
– Capable of complex translation, such as domain-specific optimizations.

In Section 2 we provide some background on the tools we will be making use of in pursuit of this goal. In Section 2.1 we survey some of the other tools for implementing domain-specific languages. In Section 3 we propose the work we plan for this thesis. In Section 3.1 we outline work beyond the scope of this thesis.

2 Background

The first major obstacle to supporting composable language extension is to allow composition of syntax extensions. Although context-free grammars are easily composed, the resulting composition may no longer be deterministic, or otherwise amenable to parser generation. Copper [15, 20] is an LR(1) parser generator that supports syntax composition of the form $H \triangleleft (E_1 \uplus_\emptyset E_2)$ so long as each $H \triangleleft E$ individually satisfy some conditions of its *modular determinism analysis*. Assuming we require extensions to satisfy this analysis, Copper offers one solution to the syntax side of the problem of supporting composable language extension.

Attribute grammars [13] are a formalism for describing computations over trees. Trees formed from an underlying context-free grammar are attributed with synthesized and inherited attributes, allowing information to flow, respectively, up and down the tree. Each production in the grammar specifies equations that define the synthesized attributes on its corresponding nodes in the tree, as well as the inherited attributes on the children of those nodes. These equations defining the value of an attribute on a node may depend on the values of other attributes on itself and its children. Attribute grammars trivially support both the "language extension" and "language unification" modes of language composition, by simply aggregating declarations of nonterminals, productions, attributes, and semantic equations.

There is a natural conflict between introducing new syntax and static analysis, referred to as the "expression problem[1]." Although normally formulated in

[1] `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`

terms of data types, it applies equally well to abstract syntax trees, and thus has consequences for language extension. If one language extension introduces new syntax, and another a new analysis, the combination of the two extensions would be missing the implementation of this analysis for this syntax. Either the composition is then broken, glue code must be written to bridge this conflict, or there must be some mechanism to accurately and automatically generate this glue code.

Attribute grammars are capable of solving the expression problem by manually providing "glue code" that provides for evaluating new attributes on new productions. However, the expression problem can also be automatically resolved *without glue code* for attribute grammars that include *forwarding* [19]. An extension production that *forwards* to a "semantically equivalent" tree in the host language can evaluate new attributes introduced in other extensions via that host language tree, where the attribute will have defining semantic equations.

Although forwarding removes the need for the "glue code" necessary to resolve the expression problem, there are other ways in which a composition of attribute grammars may cause conflicts. Attribute grammars have a "well-definedness" property that, roughly speaking, ensures each attribute can actually be evaluated. However, although H, $H \lhd E_1$ and $H \lhd E_2$ may be well-defined, there is no guarantee that $H \lhd (E_1 \uplus_{\emptyset} E_2)$ will also be well-defined. As part of this thesis, we have developed a modular well-definedness analysis [11] that provides this guarantee. This analysis checks each $H \lhd E$ individually, and ensures that the composition $H \lhd (E_1 \uplus_{\emptyset} E_2)$ will be well-defined.

2.1 Related work

Domain-specific languages are traditionally implemented as an "external" DSL, and therefore incapable of composition with each other. *Internal* (or *Embedded* DSLs) are those implemented as a "mere" library in a suitable host language [9]. Internal DSLs are interesting in part because they permit the kind of composition we are interested in. However, they come with many drawbacks. For one, not all languages are practical choices for internal DSLs, including many that are in popular use, because the range of possible syntax is seriously limited by the host language. Further, in their simplest form, internal DSLs cannot easily perform domain-specific analysis, or complex translation.

One way of making internal DSLs capable of domain-specific analysis is to take advantage of complex embeddings into the host language's type system. AspectAG [21] and Ur/Web [2] are internal DSLs that take this approach to enforcing certain properties. The drawback to these approaches is the error messages: they are reported as type errors in the host language's interpretation of the types. In the worst case, understanding these error messages requires not just a deep understanding of the property being checked, but also the particular implementation and embedding of that property into the host language's type system.

One way to improve the ability of internal DSLs to generate code is to take advantage of meta-programming facilities in the language, like LISP macros, or

C++ templates. Racket [17] offers sophisticated forms of macros to enable this kind of translation. However, the static analysis capabilities of these macros are quite limited, though they are able to generate surprisingly good error messages for a macro system. (Especially surprising for those used to C++ template error messages.)

There are several systems for specifying languages that enable language extension and unification, as described in the introduction. JastAdd [7, 4], Kiama [16], and UUAG [1] are such systems based upon attribute grammars. SugarJ [6] is a recent system built upon SDF [8] and Stratego [22]. Rascal [12] is a meta-programming language with numerous high-level constructs for analyzing and manipulating programs. Helvetia [14] is a dynamic language based upon Smalltalk with language extension capabilities. However each of these systems requires that the composition of multiple language extensions may need to be repaired with glue code, and they otherwise provide little guarantee the composition will work. As a result, they do not support composable language extension, in our sense.

MPS [23] is a meta-programming environment that leans heavily on an object-oriented view of abstract syntax, and consequently struggles with expression problem in its support for composition. Consequently, the host language limits the possible analyses over syntax that are possible. Many useful language extensions do not necessarily need new analysis over the host language, however, as macro systems for dynamic languages already demonstrate.

3 Proposal

One component of this thesis has already been mentioned: our modular well-definedness analysis for attribute grammars [11]. This work is fully described elsewhere, but we will summarize it here. We say that an attribute grammar is *effectively complete* if, during attribute evaluation, no attribute is ever demanded that lacks a defining semantic equation. This analysis operates on each $H \triangleleft E$ individually, and provides an assurances that the resulting $H \triangleleft (E_1 \uplus_\emptyset E_2)$ will also have this property, without the need to explicitly check this composed language. To do this, the analysis is necessarily conservative about what extensions pass. Roughly speaking, extensions must satisfy the following requirements:

– Extensions must not alter the *flow types* of host language synthesized attributes. That is, they cannot require new (extension) inherited attributes be supplied in order to evaluate existing (host language) synthesized attributes.
– New productions introduced in extensions must *forward.*
– The flow types for new attributes introduced by an extension must account for the potential need to evaluate forward equations before they can be evaluated.

This modular well-definedness analysis, together with Copper's modular determinism analysis, offers a potential path towards composable language extension. Silver [18, 10] is an attribute grammar-based language with support for Copper, for which we have implemented our modular well-definedness analysis.

As the remainder of this thesis, we propose to evaluate whether this tool is truly capable of composable language extension. This is not a given, because the range of potential language extensions has been restricted:

– Forwarding requires all extensions' dynamic semantics be expressible in terms of the host language. We do not anticipate this restriction being a burden, as the host languages we're interested in extending are Turing-complete with rich IO semantics.
– Copper's analysis places restrictions on the syntax that can be introduced by extensions, relative to their host language. Again, since the host languages we are interested in extending often have highly complex concrete syntax already, we expect these restrictions will be a light burden.
– Silver's analysis places restrictions on how information can flow around abstract syntax trees. Again, however, this is relative to the host language implementation, which we expect to offer support for rich kinds of information flow already.

In light of these potential restrictions on the kinds of extensions that can be specified in Silver, we wish to validate each of our goals:

– The analyses themselves accomplish the goal of supporting composable language extension.
– We will need to implement at least two new extensions to the syntax.
– We will need to implement at least one new extension to static analysis.
– That static analysis extension should demonstrate the ability to generate good, domain-specific error messages.
– One of the extensions should involve either complex translation, require domain-specific optimizations, or have at least stringent efficiency requirements, to demonstrate the approach has little to no runtime overhead.

We propose to build a host language specification for C in Silver. C is an ambitious choice, but choosing a rich, practical language of independent design is necessary to evaluate whether the analyses' restrictions are practical, as they depend on the host language. To this specification of C, we propose to build language extensions that will meet the above requirements. These should ideally be language extensions that already exist in the literature, so that the changes to their design or syntax that are necessary to satisfy the analyses can be evaluated.

From this we hope to learn:

– How to better design extensible host language implementations, to support the development of interesting extensions. Many of the limitations imposed by the analyses depend upon the host language *implementation* more so than on the host language itself.
– Ways in which Silver itself may need to be extended to help specify the host language and extensions. For example, proper aggregation of error messages in the extensions could be ensured with language features specific to error message aggregation.

– Whether the restrictions still permit interesting and practical language extensions.
– Informally, whether the resulting extended languages are useful. We intend for our colleagues to make use of these extended languages, providing some feedback in this area, though we do not intend to perform an empirical investigation.

3.1 Future work

Beyond the scope of this thesis, there lie many more problems that must be solved to bring language extension to practicality.

First, host languages must be developed in Silver before they can be extended, and extensions can only be composed for a common host language, so fragmentation must be kept to a minimum to avoid splitting apart the ecosystem. High enough quality implementations of host languages for production use remains future work.

Second, numerous less daunting engineering issues would also need resolving. No obstacles to composing language extensions at runtime exist for Silver and Copper, but the feature has yet to be fully implemented. Further, the build process for making use of an extended compiler in large software projects must be worked out.

Third, a variety of other tooling must also be composable. Languages extensions must result not only in composed compilers, but also composed debuggers and integrated development environments. Abandoning these tools is not an option for practical use. We do not intend to directly address this problem in this thesis, though concurrent work for such tools in Silver is ongoing.

Finally, although these analyses ensure conflicts do not arise from the parser or attribute evaluator, it is possible that conflicts could arise in some other fashion. Certainly we can imagine blatantly wrong code, like suppressing all error messages from subtrees. But the formulation of *composable proofs* of the compiler's correctness would complete our understanding the problem posed by composable language extension.

4 Conclusion

We believe that no existing DSL implementation tool satisfies all five goals listed in the introduction: support composable language extension, allow extension to both and static analysis, provide good domain-specific error messages, and allow complex translation requirements. These goals are motivated by the desire to ensure that the users of language extensions can be certain they can draw on whatever high-quality extensions they need, without fear of breaking their compiler.

We have developed an analysis that ensures Silver meets the goal of supporting composable language extension, and we have implemented this analysis. We intend to develop an extensible specification of a popular and practical language, C, and we intended to demonstrate that practical language extensions to it are possible that satisfy this analysis. We believe this will demonstrate that Silver satisfies all five goals listed in the introduction for an ideal DSL implementation technique.

References

1. Baars, A., Swierstra, D., Loh, A.: Utrecht University AG system manual, http://www.cs.uu.nl/wiki/Center/AttributeGrammarSystem.
2. Chlipala, A.: Ur: statically-typed metaprogramming with type-level record computation. In: PLDI, 2010. pp. 122–133. ACM, New York, NY, USA (2010), `http://doi.acm.org/10.1145/1806596.1806612`
3. Deursen, A.v., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. ACM SIGPLAN Notices 35(6), 26–36 (Jun 2000)
4. Ekman, T., Hedin, G.: Rewritable reference attributed grammars. In: Proc. of ECOOP '04 Conf. pp. 144–169 (2004)
5. Erdweg, S., Giarrusso, P., Rendel, T.: Language composition untangled. In: LDTA, 2012 (2012)
6. Erdweg, S., Rendel, T., Kästner, C., Ostermann, K.: Sugarj: library-based syntactic language extensibility. In: OOPSLA 2011. pp. 391–406. ACM, New York, NY, USA (2011), `http://doi.acm.org/10.1145/2048066.2048099`
7. Hedin, G., Magnusson, E.: JastAdd - an aspect oriented compiler construction system. Science of Computer Programming 47(1), 37–58 (2003)
8. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism sdf. SIGPLAN Not. 24(11), 43–75 (Nov 1989), `http://doi.acm.org/10.1145/71605.71607`
9. Hudak, P.: Building domain-specific embedded languages. ACM Computing Surveys 28(4es) (1996)
10. Kaminski, T., Van Wyk, E.: Integrating attribute grammar and functional programming language features. In: Proceedings of 4th the International Conference on Software Language Engineering (SLE 2011). LNCS, vol. 6940, pp. 263–282. Springer (July 2011)
11. Kaminski, T., Van Wyk, E.: Modular well-definedness analysis for attribute grammars (2012), accepted SLE 2012
12. Klint, P., van der Storm, T., Vinju, J.: Rascal: a domain specific language for source code analysis and manipulation. In: Proc. of Source Code Analysis and Manipulation (SCAM 2009) (2009)
13. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory 2(2), 127–145 (1968), corrections in **5**(1971) pp. 95–96
14. Renggli, L., Gîrba, T., Nierstrasz, O.: Embedding languages without breaking tools. In: ECOOP 2010. pp. 380–404. Springer (2010)
15. Schwerdfeger, A., Van Wyk, E.: Verifiable composition of deterministic grammars. In: Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). ACM Press (June 2009)
16. Sloane, A.M.: Lightweight language processing in kiama. In: Proc. of the 3rd summer school on Generative and transformational techniques in software engineering III (GTTSE 09). pp. 408–425. Springer (2011)
17. Tobin-Hochstadt, S., St-Amour, V., Culpepper, R., Flatt, M., Felleisen, M.: Languages as libraries. In: PLDI 2011. pp. 132–141. ACM, New York, NY, USA (2011), `http://doi.acm.org/10.1145/1993498.1993514`

18. Van Wyk, E., Bodin, D., Krishnan, L., Gao, J.: Silver: an extensible attribute grammar system. Scinece of Computer Programming (2009), accpeted, In Press
19. Van Wyk, E., de Moor, O., Backhouse, K., Kwiatkowski, P.: Forwarding in attribute grammars for modular language design. In: Proc. 11th Intl. Conf. on Compiler Construction. LNCS, vol. 2304, pp. 128–142 (2002)
20. Van Wyk, E., Schwerdfeger, A.: Context-aware scanning for parsing extensible languages. In: Intl. Conf. on Generative Programming and Component Engineering, (GPCE). ACM Press (October 2007)
21. Viera, M., Swierstra, S.D., Swierstra, W.: Attribute grammars fly first-class: How to do aspect oriented programming in haskell. In: Proc. of 2009 International Conference on Functional Programming (ICFP'09) (2009)
22. Visser, E.: Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In: Rewriting Techniques and Applications (RTA'01). LNCS, vol. 2051, pp. 357–361. Springer-Verlag (2001)
23. Voelter, M.: Language and ide modularization, extension and composition with mps. In: GTTSE 2011 (2011)