Migrating Custom DSL Implementations to a Language Workbench (Tool Demo)

Jasper Denkers
Delft University of Technology
Delft, The Netherlands
j.denkers@tudelft.nl

Louis van Gool
Océ Technologies B.V.
Venlo, The Netherlands
louis.vangool@oce.com

Eelco Visser
Delft University of Technology
Delft, The Netherlands
e.visser@tudelft.nl

Abstract

We present a tool architecture that supports migrating custom domain-specific language (DSL) implementations to a language workbench. We demonstrate an implementation of this architecture for models in the domains of defining component interfaces (IDL) and modeling system behavior (OIL) which are developed and used at a digital printer manufacturing company. Increasing complexity and the lack of DSL syntax and IDE support for existing implementations in Python based on XML syntax hindered their evolution and adoption. A reimplementation in Spoofax using modular language definition enables composition between IDL and OIL and introduces more concise DSL syntax and IDE support. The presented tool supports migrating to new implementations while being backward compatible with existing syntax and related tooling.

CCS Concepts • **Software and its engineering → Domain specific languages**;

Keywords domain-specific languages, migration

ACM Reference Format:
Jasper Denkers, Louis van Gool, and Eelco Visser. 2018. Migrating Custom DSL Implementations to a Language Workbench (Tool Demo). In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (SLE '18), November 5–6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3276604.3276608

1 Introduction

Organizations developing software benefit from model-driven development by introducing abstractions. Such abstractions reduce boilerplate code and implementation complexity. Additionally, they improve quality and developer effectiveness. The digital printer manufacturing company Océ applies model-based development for such reasons. First, Interface Definition Language (IDL) to define component interfaces and to automatically generate interface code that connects several target languages and that provides standardized logging of all interface communication. Second, Océ Interaction Language (OIL) to specify, analyze, and implement system behavior using state machines. Existing implementations for these models consist of XML syntax based on XSD schemas and Python scripts that perform analysis and code generation, packaged in command-line and web-based utilities. Successful application for over 15 years proved IDL to be a viable approach. The more complex OIL is under active development and used in production software at a limited scale only.

Continuous development and maintenance of custom model implementations with well-known technologies is flexible, but involves much boilerplate code and complexity grows while implementations evolve. Still, developing tailor-made implementations even for possibly non-unique domains is justified, especially in an industrial setting where e.g. integration and usage circumstances are unique. However, providing a good environment for developers to work with such models, including concise DSL syntax and interactive IDE support, is expensive. Therefore, organizations need better tools for their DSL implementations and maintenance. While evolving to better tools, the new implementations need to be backward compatible with existing implementations and related tooling, and ideally provide migration solutions for porting legacy code.

Language workbenches [3–5] are tools for implementing DSLs and automatically provide IDEs. Spoofax [6, 8] is a particular instance for developing textual DSLs. This work presents an industrial tool migration of IDL and OIL to implementations in Spoofax. First, we present a language architecture that supports accepting multiple syntaxes simultaneously. We implement this architecture for IDL and OIL and demonstrate how it enables transforming between XML and DSL syntax. This enables automated migration between existing XML programs and the new DSL format. Additionally, new programs written in the DSL syntax are backward compatible with existing tooling based on XML

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SLE '18, November 5–6, 2018, Boston, MA, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6029-6/18/11…$15.00
https://doi.org/10.1145/3276604.3276608

syntax. Second, we demonstrate how Spoofax supports implementing this architecture using modular syntax definition, transformations, and cross-language reference resolution.

2 IDL and OIL

This section describes the languages IDL and OIL and their existing implementations.

2.1 IDL: Component Interfaces

IDL serves as a language-agnostic and modular method to define types with constraints and interfaces based on such types. From IDL definitions, code generators can generate interfaces in several target languages like C++ and C#. Additionally, generated code enforces the constraints on types and logs all interface communication.

Modules are the top level units in IDL. By using *imports*, one module can reuse types of another, thus enabling modular definitions. *Types* define either named aliases of standard primitive types such as numbers, strings, and lists, or define instances of enums or structs. Types can optionally define *constraints* that restrict their possible values. Such constraints are boolean expressions and translate to assertions in constructors of generated code. IDL definitions contain *interfaces* similar to those in languages such as Java or C#. Interfaces contain a set of *methods* which define *parameters* and *return* values, typed by primitives or the custom types defined elsewhere in IDL.

2.2 OIL: System Behavior

OIL is a language for the specification, analysis, and implementation of system behavior. It presents a layer on top of IDL in which IDL interfaces define the communication channels between components. Method calls and replies on these interfaces represent events on the channels, triggering state machines that model protocol specifications or component behavior. Based on logging of events that pass the interfaces, developers can analyze behavior in retrospect. Additionally, OIL component specifications can automatically generate fully functional code.

2.3 Existing Implementations

The existing implementations for both IDL and OIL follow the same recipe. First, an XSD schema defines a subset of XML as the model syntax. Second, Python scripts implement parsing (using an existing XML parsing library), static analysis (using AST traversals and ad-hoc name binding and type analysis) and code generation. Several tools operate on the XML syntax, e.g. for separate (company-project specific) code generation or web-based visualization of OIL state machines.

The existing implementations are tailored to code generation and do not provide interactive IDE functionality. Python scripts implement heuristics to perform static checks. Recursive traversal of expression ASTs gathers type information bottom up and checks references and well-formedness. Developers edit programs in their editor of choice and run the scripts via the command line or web application to get analysis feedback as console output. Build systems include the code generation scripts. Code generators do not check the correctness of programs and thus incorrect programs can lead to incorrect generated code without being noticed by developers.

2.4 Analysis

Several problems arise with the existing implementations of IDL and OIL. Both languages lack concise DSL syntax and interactive IDE support. This causes inconvenient editing of programs in XML syntax. Analysis results are only reported after manually triggering scripts, whose output needs to be manually traced back to the specification. The development of OIL is hindered due to increasing complexity since e.g. no trivial method for advanced features like cross-language reference resolution are available. Therefore, migrating the implementation to a language workbench is useful. Higher levels of abstractions for implementing language components help reduce complexity, and automatic derivation of IDE support is provided for free.

Re-implementing DSLs in a language workbench leads to several requirements regarding migration. First, when introducing a new syntax, programs in this syntax must be backward migratable to prevent having to update all existing tooling. Additionally, existing programs must be forward migratable to the new syntax. Finally, analysis and generated code must be easily integratable in the build system and developers' workflow.

2.5 Approach

We focus on two aspects of migrating IDL and OIL. First, we design a concise DSL syntax for the languages (Section 3). Second, we define the architecture that support the migration (Section 4). The work is performed in Spoofax [6, 8], the integration of several declarative meta-DSLs from which we use SDF3 for syntax definition, NaBL2 for static analysis based on scope graphs [1, 7] and Stratego for transformations [2]).

3 Language Design

This section describes and motivates the design of the introduced DSL syntaxes for IDL and OIL. The objective is to provide DSLs that are more concise and readable, that are tailored towards domain concepts, and that do not have the overhead XML syntax has. For IDL, the DSL syntax could follow naturally from the concepts and hierarchy of the XML schema. The new DSL mostly focuses on removing boilerplate syntax. For OIL the specification of the DSL was less straightforward. Especially for transitions a more natural

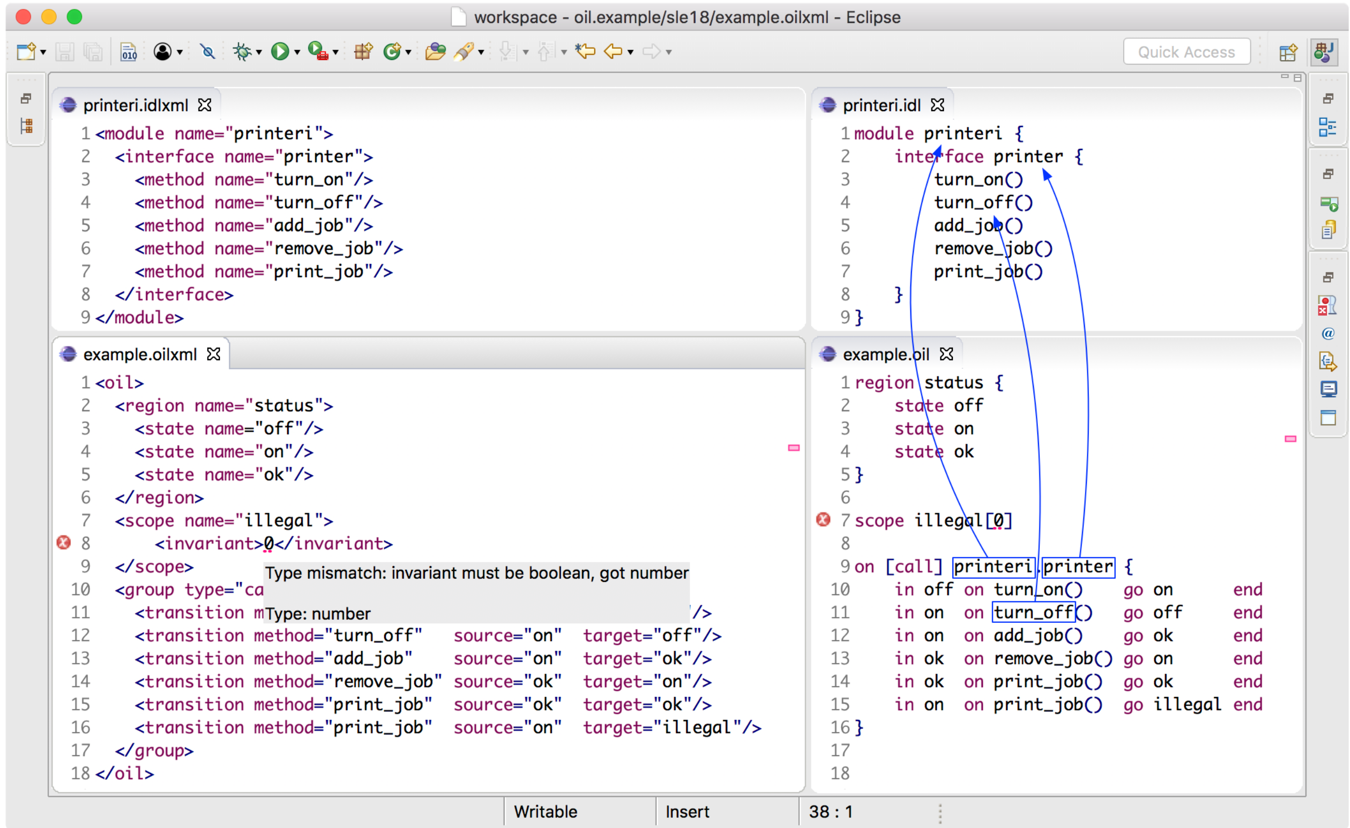

Figure 1. An example OIL program in the Eclipse environment of Spoofax. The left program is written in the existing XML syntax. The right program is the same program migrated to the DSL syntax. The blue arrows indicate cross-language reference resolution. Errors are reported interactively to the user.

reading was possible. To make the transition to the new syntax easier for developers, the syntax looks similar to well-known languages like C and Java by using curly braces. For consistency, definitions follow the pattern of a type followed by a name and then optionally additional information between curly braces. References to types start with a name, then a colon and the type. Figure 1 contains an example IDL and OIL program in XML and its translation to the DSL syntax. Table 1 indicates the conciseness of the DSLs by comparison with their XML counterparts.

Table 1. Comparison of XML and DSL file size for the biggest IDL with and without constraints and OIL programs present. Sizes are in lines of code (and number of characters).

	XML	DSL	Δ
IDL without constraints	14799 (583586)	12276 (324344)	17% (44%)
IDL with constraints	643 (25374)	491 (12761)	24% (50%)
OIL	747 (48204)	374 (31084)	50% (36%)

4 Migration Architecture

This section describes the implementation architecture of IDL and OIL in Spoofax that enables language composition and supports migrations.

4.1 Language Pipeline

For both IDL and OIL the same language pipeline architecture is applied. The key characteristic of this architecture is the introduction of a normalized AST format. Bi-directional transformations from both syntaxes to the normalized AST enable single definition of static analysis and code generation that is applicable on both the XML and DSL programs. Additionally, this automatically derives support for migrating old programs to the better readable DSL, and new programs written in the DSL stay backward compatible with existing tools since they can be transformed to the XML syntax. Figure 2 visualizes this pipeline used for both languages.

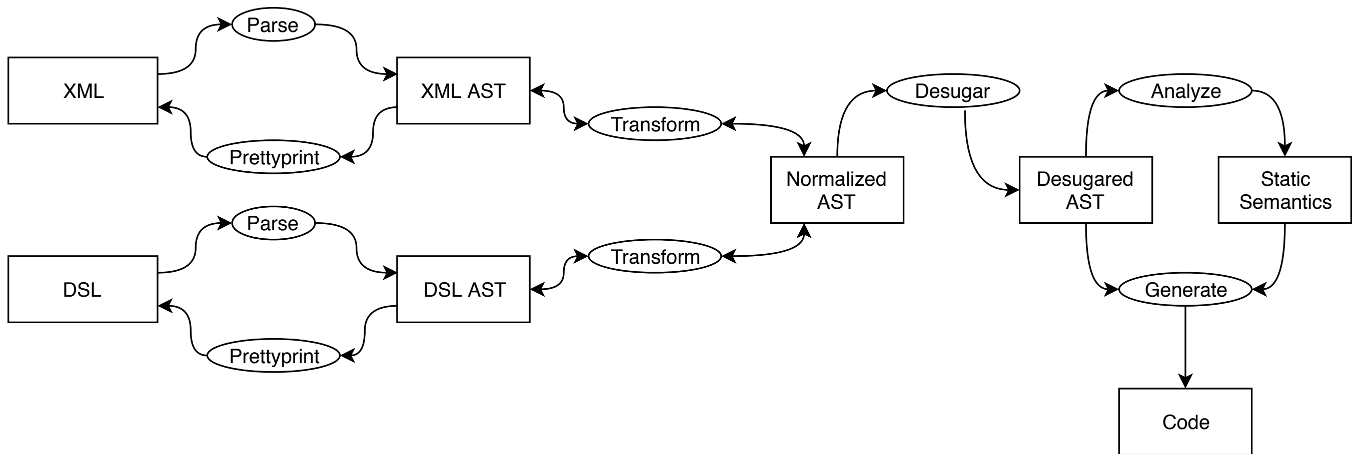

Figure 2. The language pipeline implemented for both IDL and OIL. Rectangles indicate model instance components and ellipses indicate transformations between them. SDF3 definitions generate parsers and pretty printers. Stratego definitions implement the transformations (XML/DSL to and from normalized AST, desugaring, and code generation). NaBL2 definitions generate static analysis code. Code generation uses static analysis results by e.g. taking type information into account.

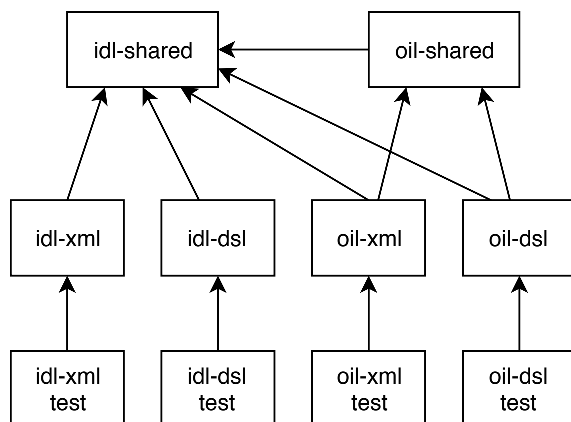

Figure 3. The Spoofax projects hierarchy for the implementation of IDL and OIL, including separate definitions of shared code and language variants for XML and DSL syntax. Arrows indicate dependencies between projects.

4.2 Project Hierarchy

In total four Spoofax language projects implement the two languages (IDL and OIL) in two formats (XML and DSL). Additionally, each instantiation has a separate project for testing. Since the two formats per language have a single definition of static semantics and code generation, both languages have a parent project with shared definitions on which the two formats depend. Figure 3 shows the hierarchy of projects and their dependency relationships. OIL reuses syntax, static semantics and AST signatures definitions for keywords, expressions, and types from IDL, justifying the dependency from OIL projects on the IDL shared project.

4.3 Composition

Composition between IDL and OIL happens on the level of syntax and static analysis. OIL imports reuses syntax definition of expressions from IDL. Events in OIL correspond to methods defined in IDL. We need cross-language reference resolving to statically check whether the events used in OIL are valid. Static analysis definition with NaBL2 is based on scope graphs. We realize the composition of static analysis by merging the two languages and their scope graphs, thus enabling resolution in IDL modules from OIL. This is a workaround; ideally Spoofax and NaBL2 would support importing the root scope graph node from one language to another.

5 Conclusions

We presented an architecture for implementing DSLs in a language workbench that supports migrations from old to new syntax and vice versa. The key characteristic of this architecture is an intermediate normalized AST with bi-directional transformations to and from both syntaxes. We implemented an instance of this architecture for IDL and OIL in Spoofax. It demonstrates how the introduction of improved DSLs with concise syntax and IDE support can be adopted while being backward compatible with old syntax. This lowers the boundary for industry to adopt language workbenches for custom DSL implementations.

Acknowledgements

This research was funded by a grant from the Top Consortia for Knowledge and Innovation (TKIs) of the Dutch Ministry of Economic Affairs and from Océ. The authors would like to thank Olav Bunte and the anonymous reviewers for their feedback.

References

[1] Hendrik van Antwerpen, Pierre Néron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. 2016. A constraint language for static semantic analysis based on scope graphs. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. ACM, 49–60.

[2] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2008. Stratego/XT 0.17. A language and toolset for program transformation. *Science of computer programming* 72, 1-2 (2008), 52–70.

[3] Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. 2013. The state of the art in language workbenches. In *International Conference on Software Language Engineering*. Springer, 197–217.

[4] Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures* 44 (2015), 24–47.

[5] Martin Fowler. 2005. Language workbenches: The killer-app for domain specific languages. (2005).

[6] Lennart CL Kats and Eelco Visser. 2010. The spoofax language workbench: rules for declarative specification of languages and IDEs. In *ACM sigplan notices*, Vol. 45. ACM, 444–463.

[7] Pierre Neron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. 2015. A theory of name resolution. In *European Symposium on Programming Languages and Systems*. Springer, 205–231.

[8] Eelco Visser, Guido Wachsmuth, Andrew Tolmach, Pierre Neron, Vlad Vergu, Augusto Passalaqua, and Gabriël Konat. 2014. A Language Designer's Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. ACM, 95–111.