Software and Systems Modeling (2025) 24:139–182
https://doi.org/10.1007/s10270-024-01185-x

REGULAR PAPER

OIL: an industrial case study in language engineering with Spoofax

Olav Bunte[1] · Jasper Denkers[2] · Louis C. M. van Gool[3] · Jurgen J. Vinju[1,4] · Eelco Visser[2] · Tim A. C. Willemse[1] · Andy Zaidman[2]

Received: 15 September 2023 / Revised: 26 March 2024 / Accepted: 8 May 2024 / Published online: 3 June 2024

Abstract
Domain-specific languages (DSLs) promise to improve the software engineering process, e.g., by reducing software development and maintenance effort and by improving communication, and are therefore seeing increased use in industry. To support the creation and deployment of DSLs, language workbenches have been developed. However, little is published about the actual added value of a language workbench in an industrial setting, compared to not using a language workbench. In this paper, we evaluate the productivity of using the Spoofax language workbench by comparing two implementations of an industrial DSL, one in Spoofax and one in Python, that already existed before the evaluation. The subject is the Open Interaction Language (OIL): a complex DSL for implementing control software with requirements imposed by its industrial context at Canon Production Printing. Our findings indicate that it is more productive to implement OIL using Spoofax compared to using Python, especially if editor services are desired. Although Spoofax was sufficient to implement OIL, we find that Spoofax should especially improve on practical aspects to increase its adoptability in industry.

Keywords Language workbench · Language engineering · Case study

1 Introduction

Every piece of software is written in one or more software languages. The most common software languages are general-purpose languages (GPLs), such as C++, Java, and Python. For specific purposes, it can be beneficial to design a tailored language. Such a language is called a *domain-specific language* [1] (DSL). Compared to GPLs, DSLs promise to improve the software engineering process, e.g., by reducing development and maintenance effort when implementing (domain-specific) software. They are also considered to be more suitable for communication between software engineers and domain experts [2].

To support the creation and deployment of DSLs, *language workbenches* have been developed [3–5]. Language workbenches are specifically designed for the development of a DSL. This includes the DSL's syntax, from which parsers can be derived automatically, as well as its semantics, e.g., by means of a translation to other languages. Language workbenches typically also generate IDEs for the DSLs implemented in them. Examples of language workbenches are MPS [6], Xtext [7], Rascal [8], and Spoofax [9].

Although there already is ample literature on the underlying theory of language workbenches (e.g., [5, 10]), little is

Communicated by Juan de Lara.

Olav Bunte and Jasper Denkers have contributed equally to this work.

✉ Olav Bunte
o.bunte@tue.nl

✉ Jasper Denkers
j.denkers@tudelft.nl

Louis C. M. van Gool
louis.vangool@cpp.canon

Jurgen J. Vinju
jurgen.vinju@cwi.nl

Eelco Visser
e.visser@tudelft.nl

Tim A. C. Willemse
t.a.c.willemse@tue.nl

Andy Zaidman
a.e.zaidman@tudelft.nl

1 Eindhoven University of Technology, Eindhoven, The Netherlands

2 Delft University of Technology, Delft, The Netherlands

3 Canon Production Printing, Venlo, The Netherlands

4 CWI, Amsterdam, The Netherlands

documented about the actual added value of language workbenches compared to not using language workbenches in an industrial setting when designing and engineering DSLs. This is relevant for two main reasons. On the one hand there is opportunity: there exist DSL implementations in industry which have not been developed with the potential benefit of language workbenches. On the other hand, there are still unknowns: most language workbenches spawn from academic environments which can have different views on software engineering effectiveness compared to a pure industrial setting. How relevant are the benefits of language workbenches in an industrial setting?

One of the first works that evaluates the added value of a specific language workbench in an industrial context is the work by Van den Brand et al. [11]. In this work, the authors present some experiences with using ASF+SDF for railway and financial domains. A more recent and extensive industrial case study is described in the work of Voelter et al. [12]. The authors evaluate the MPS language workbench with as case the mbeddr collection of languages under non-trivial requirements. The work by Voelter et al. resulted in meaningful lessons learned for the particular case study from an industrial perspective. Still, the authors call for more studies on language workbench evaluation to expand our knowledge of the usefulness of language workbenches for language engineering in general. This will help industrial language engineers decide when and how to use language workbenches.

We present such an evaluation of the Spoofax language workbench in an industrial setting. In the original work on Spoofax [9], the authors claim that Spoofax "*enables efficient, agile development of software languages with state-of-the-art IDE support based on concise, declarative specifications*". From this it can be derived that it should be more productive to implement a DSL with Spoofax compared to when not using a language workbench. Although it has been demonstrated that Spoofax is able to deliver on its original promises for non-industrial greenfield situations, e.g., in the area of web programming [13, 14], or declarative data modeling [15, 16], it is unclear to what extent the original claims of Spoofax still hold for our industrial case. This leads to the following research question:

RQ: *How does the productivity of implementing an industrial language in Spoofax compare to the productivity when using a GPL and available libraries?*

Productivity is about the amount of effort needed to implement some functionality. As a proxy for effort we measure code volume, as it is the only information available to us in this study that relates to effort and can be measured objectively.

The industrial case with which we evaluate Spoofax is the Open Interaction Language (OIL), a textual language for modeling control software, developed at Canon Production Printing.[1] Before the implementation of OIL in Spoofax was created, a design of OIL already existed based on XML, along with an implementation in Python. This makes OIL a typical industrial case, in the sense that the new implementation must fit into an existing software ecosystem which is used to create commercial products.

The industrial context requires a number of features for the implementation of OIL. In particular, with the migration to Spoofax, the original XML syntax should be supported alongside a new more user-friendly syntax. OIL's syntax allows the user to leave out boilerplate information, which the implementation needs to make explicit. The well-formedness, name binding, and typing of an OIL specification should be statically checked and errors should be reported to the user. OIL specifications depend on modules and interfaces defined in another language called Interface Definition Language (IDL). Finally, the Spoofax implementation of OIL should be able to generate code, both for the execution and for the verification of OIL specifications.

Based on the aforementioned requirements and earlier non-industrial evaluations of Spoofax, in this paper we evaluate how well Spoofax can cope with the complexity and scale of the industrial OIL case study. The development of OIL in Spoofax, executed by five developers over more than four years, allows us to make interesting observations on language engineering, distill strengths and weaknesses of Spoofax, derive lessons learned for future language engineering efforts, and propose areas of future work to improve the language workbench.

1.1 Outline

This paper is structured as follows: First we provide background on Spoofax in Sect. 2 and on OIL in Sect. 3. We dive into the context and setup of the case study and we elaborate on our research question in Sect. 4. Next, we discuss the language engineering aspects of OIL's implementation in Spoofax in separate sections, and we evaluate our research question for each aspect at the end of those sections. We discuss the implementation of OIL's concrete syntax in Sect. 5. In Sect. 6, we discuss the abstract syntax representation of OIL. Then in Sects. 7 and 8, we discuss the implementation of the static and dynamic semantics of OIL, respectively. In Sect. 9, we summarize our findings regarding the research question and discuss threats to validity. We discuss experiences that are not directly related to the research question in Sect. 10, as well as list our lessons learned and provide a research agenda for Spoofax. We position our work with that of others in Sect. 11. We conclude in Sect. 12.

[1] https://cpp.canon.

2 Spoofax

In this section, we provide background information on Spoofax, which is useful for understanding the way that OIL is implemented in Spoofax in later sections. Spoofax[2] is an open source language workbench that promises to support the development of textual DSLs by offering meta-DSLs (DSLs for specifying DSLs) for concise, declarative specifications of languages and IDE services [17]. The idea of declarative language definition is that language developers focus on the high-level specification of their languages rather than focusing on the low-level implementation of, e.g., parsing or type checking algorithms. Based on language aspects specifications in the meta-DSLs, Spoofax automatically generates an IDE.

Spoofax is developed at the Delft University of Technology since 2007 [9], building on previous work on syntax definition with SDF2 [18] and program transformation with the Stratego XT toolset [19]. Besides SDF2 and Stratego, the first version of Spoofax offered meta-DSLs for static semantics (NaBL2 [20]), editor services (ESV), and testing (SPT). Spoofax Core [21], implemented in Java, integrates the meta-DSLs and provides a build system to automatically transform language specifications into implementations. Spoofax is primarily deployed as a plugin for the Eclipse IDE.[3]

Developments on Spoofax since the introduction of the language workbench include:

- **Syntax.** The syntax definition formalism SDF3 [22] with support for template-based syntax definition.
- **Transformation.** The program transformation language Stratego 2 with support for gradual typing [23] and incremental builds [24].
- **Static Semantics.** Static semantics specification (NaBL2 and its successor Statix [25], both based on a scope graph model [26]) and support for incremental type checking [27].
- **Data-Flow.** The data-flow analysis specification language FlowSpec [28].
- **Incremental Builds.** Interactive software development pipelines with PIE [29].
- **IDE support.** Static semantic code completion [30].
- **Testing** Language test suites with the Spoofax Testing language (SPT).

The case study in this paper has been performed with Spoofax version 2 [21], making use of the SDF3, NaBL2, Stratego, ESV, and SPT meta-DSLs. Spoofax version 3 (including Stratego 2, Statix, and PIE) was under development during the execution of this study and could therefore not yet be considered. In Sect. 10, we discuss how our findings relate to Spoofax 3.

```
let x = 20 + 1 in 2 * x
```

```
Exp(
  Let(
    "x"
  , Add(Int("20"), Int("1"))
  , Mul(Int("2"), Ref("x"))
  )
)
```

Fig. 1 An example program in EXP and its corresponding abstract syntax in ATerm

In the remainder of this section, we discuss both conceptual and practical aspects of language engineering with Spoofax, which are important for understanding the Spoofax implementation of OIL. Also, we will further introduce the meta-DSLs for the language aspects that are relevant in our case study. We use a simple expressions language EXP as a running example, which supports integers, addition, multiplication, let bindings and references. Figure 1 depicts an example EXP program and its corresponding abstract syntax.

2.1 Anatomy of Spoofax projects

A Spoofax project consists of source files and configuration files. The source files primarily consist of specifications in the meta-DSLs. For integrating a language implementation with external libraries, Java source files can be included as well. The language build and dependencies are configured in the configuration files. All sources files are textual and are therefore typically stored in a version control system.

Based on the source files and configuration files, Spoofax generates language artifacts such as parse tables, AST schemas, and ultimately the complete language implementation in the form of an Eclipse plugin. During a language build, besides the sources that the language developer writes, additional sources are generated which can be referenced by other specifications or form an input for a further build step. For example, signatures are generated automatically from the SDF3 grammar and can be used in Stratego to define transformation rules on. These generated sources are stored separately from the main source files and are typically ignored in version control.

Projects can define a complete language, define (library) sources intended for reuse by other language projects, or only define a transformation for an existing language. Through dependencies between projects, different forms of language composition can be realized. For example, a language project can re-use definitions of another language by adding that

[2] https://spoofax.dev.

[3] https://www.eclipse.org/ide/.

```
signature
  constructors

    Int : INT -> Exp
    Add : Exp * Exp -> Exp
    Mul : Exp * Exp -> Exp
    Let : ID * Exp * Exp -> Exp
    Ref : ID -> Exp
```

Fig. 2 A Stratego code snippet that defines an AST schema for EXP

project as a dependency. Also, a project can contribute a transformation to an existing language, such that more functionalities become available for a language, independent from its original implementation.

2.2 Data representation with ATerms

The language ATerm (Annotated Terms) [31] defines the representation of abstract syntax trees (ASTs) and data used by most other meta-DSLs. These ASTs and data consist of terms (often referred to as "ATerms") that can be annotated with additional data. A term can either be a number, a string, a list of terms, or a constructor with zero or more subterms. The annotations on terms are typically used to store metadata, such as static analysis results. ATerm serves as the "glue" between the meta-DSLs. For example, the output of an SDF3-based parser is an AST expressed in ATerm. ATerm is the object language for the Stratego transformation meta-DSL, i.e., Stratego defines transformation rules for terms expressed in ATerm. Also, name binding and typing specifications in NaBL2 consist of rules that apply to ATerm patterns.

Terms must adhere to many-sorted algebraic *signature* [32] definitions, which are defined in Stratego. The signatures define *sorts* and *constructors*. Sorts represent syntactic categories (also known as non-terminals, e.g., `Exp`) and constructors specify instances of these sorts (e.g., `Add`). The snippet in Fig. 2 contains signature definitions for EXP. It defines unary `Int` and `Ref` constructors for the sort `Exp`, binary `Add` and `Mul` constructors for the sort `Exp`, and a ternary `Let` constructor. Figure 1 contains an example term that conforms to the signatures.

2.3 Syntax definition with SDF3

SDF3 [22] is a syntax definition language that covers more than realizing a parser implementation based on a grammar specification. From an SDF3 grammar, the following language implementation artifacts are generated automatically: an AST schema, a parser with error recovery, a pretty printer, a parenthesizer, syntax highlighting, and syntactic code completion. The SDF3 formalism extends context-free

```
1  module lex
2
3  lexical syntax
4
5    INT    = "-"? [0-9]+
6    ID     = [a-zA-Z] [a-zA-Z0-9\]*
7    LAYOUT = [\ \t\n\r]
```

```
1  module exp
2
3  imports lex
4
5  context-free syntax
6
7    Exp.Int = INT
8    Exp.Add = [[Exp] + [Exp]] {left}
9    Exp.Mul = [[Exp] * [Exp]] {left}
10   Exp.Let =
11     [let [ID] = [Exp] in [Exp]] {non-assoc}
12   Exp.Ref = ID
13
14 context-free priorities
15
16   Exp.Mul > Exp.Add > Exp.Let
```

Fig. 3 Two SDF3 code snippets that define the syntax for EXP

grammars [33] with high-level syntax definition features such as constructor declarations (used for AST schema generation in the form of ATerm signatures), disambiguation constructs (for disambiguation and generating a parenthesizer), and templates (for deriving pretty printers) [34].

See Fig. 3 for two modules of SDF3 that define the syntax of EXP. The second module (`exp`) imports the first module (`lex`). For example, in `exp`, the rule on line 7 defines a rule for integers, using the lexical syntax for `INT` defined in `lex`. The rules on lines 8–9 are defined to be left-associative using the `{left}` disambiguation construct. The left-hand sides of grammar rules consist of a sort (e.g., `Exp`) and optionally a constructor declaration (e.g., `.Add`). The signatures of Fig. 2 are generated automatically based on the constructor declarations in this grammar.

In addition to associativity declarations for disambiguation of an operator with itself, the `context-free priorities` section defines disambiguation through priorities between operators. In the example, `Exp.Mul` has higher priority than `Exp.Add` which has higher priority than `Exp.Let` (line 16). Priority declarations are transitive. When importing modules in SDF3, their disambiguation rules are imported with them as well. Note that this may create new ambiguities between grammar elements of different modules, so additional disambiguation rules may need to be defined.

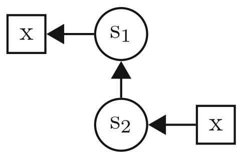

Fig. 4 The scope graph that corresponds to the example EXP program of Fig. 1. Scope s_1 is the root scope node and corresponds to the while program. Scope s_2 is the scope introduced for the body of the let operator, consisting of the expression $x = 20 + 1$. The declaration of x is represented by the outgoing edge from scope s_1. The reference of x is represented by the incoming edge to scope s_2. This name binding in this program is valid, as there exists a path from the reference of x to its declaration

SDF3-based parsing involves the process of *imploding*, i.e., transforming parse trees into ASTs. Only the nodes in the parse tree that are parsed based on grammar rules for which a constructor has been declared end up in the AST, which filters out irrelevant details of the concrete syntax such as white space and comments. This filtering is necessary because SDF3 uses a scannerless parsing approach [18, 35], a foundational characteristic which makes SDF3 grammar composable.

During parsing and imploding, the created AST terms are annotated with the origin location of the parsed syntactic element in the input program, which is the first step of *origin tracking* [36]. These origins can be maintained during transformations, which is useful for, e.g., error reporting.

To adapt the formatting of the text produced by the pretty printer that is generated from the SDF3 grammar, one can enhance the SDF3 grammar with templates. The example of Fig. 3 uses such templates for `Add`, `Mul` and `Let`, which is indicated by the square brackets that are placed around the right-hand sides of the grammar rules. Any formatting between these square brackets, including spaces, tabs, and newlines, will be used by the pretty printer. In case square brackets are part of the grammar definition, angular brackets can be used instead.

2.4 Static semantics with NaBL2

NaBL2 [20, 37], pronounced as “enable two”, is a static semantics definition language which covers name binding and type systems based on the *scope graph* model [26]. Given an NaBL2 specification, programs are transformed into constraints and a scope graph which captures the binding structure and typing of the program. Name resolution corresponds to finding a path in the graph from a reference to its declaration. An NaBL2 specification contains constraint generation rules for every term in the AST schema of the language, with conditions that specify how the term contributes to scope graph generation, name binding, and typing.

Figure 4 depicts the scope graph that corresponds to the example EXP program of Fig. 1. Each term is made part of a scope, which is a node in the scope graph. Declarations and references (e.g., of variables) are also added as nodes in the scope graph. For declarations in a scope, we add a node for the declaration with an edge from the scope to the declaration. For language constructs that introduce a deeper level in the overall scoping structure, the scope is added to the graph as a new node with an edge to the parent scope. For a reference, a node is added with an edge from the reference node to the node of the scope the reference is made from. Name resolution then boils down to finding a path in the scope graph from the reference to the corresponding declaration.

By assigning types to terms, type analysis can check or infer types. Conditions in constraint rules can be extended with an error message applied to a term. Whenever a condition fails, the error message can be displayed on the origin of the term using origin tracking.

See Fig. 5 for an NaBL2 snippet that specifies name and typing rules for EXP. Four rules are defined, identified in double square brackets (lines 2, 4, 9 and 17). The rule for term `Int` (line 2) assigns the type `TInt` to the term. The rule for `Add` recursively specifies the semantics for its subexpressions by calling constraint rules on `exp1` and `exp2` using double square brackets (lines 5–6). Note that no rule references are used: the rule that needs to be applied depends on the outermost constructor of the sub-expressions. Afterward, it is defined that their types should be the same (line 7). For more complex type systems, it is possible to define relations between types. This enables, for instance, the addition of an integer with a float and the computation of the resulting type. The rule for `Mul` has been omitted as it is similar to the rule for `Add`.

The rule for let bindings (line 9) introduces a new scope `s'` (line 10), sets `s` as the parent scope of `s'` (line 11) and attaches a declaration node `Var{x}` for name `x` in the namespace `Var` to scope `s'` using an arrow pointing toward the declaration node (`<-`, line 12). It then analyses the first expression within scope `s` (line 13) and assigns the derived type `ty1` to the declaration node (line 14). Lastly, it continues the analysis with the second expression within scope `s'` (line 15). The rule for variable references (line 17) first attaches a reference node `Var{v}` for name `v` in the namespace `Var` on scope `s` using an arrow pointing away from the reference node (`->`, line 18). Afterward, it is checked whether some declaration `d` exists for reference `Var{v}` using operator `|->` (line 19), effectively checking whether there exists a path through the scope graph from the reference node to a declaration node with the same name and namespace. We then require that this declaration `d` has type `ty` (line 20), which is the same type as the `Ref` term that the rule is defined on (line 17).

```
rules
  [[ Int(_) ^ (s) : TInt() ]].

  [[ Add(exp1, exp2) ^ (s) : ty1 ]] :=
    [[ exp1 ^ (s) : ty1 ]],
    [[ exp2 ^ (s) : ty2 ]],
    ty1 == ty2 | error $[Type mismatch:
    cannot add [ty2] to [ty1]].

  [[ Let(x, exp1, exp2) ^ (s) : ty2 ]] :=
    new s',
    s' ---> s,
    Var{x} <- s',
    [[ exp1 ^ (s) : ty1 ]],
    Var{x} : ty1,
    [[ exp2 ^ (s') : ty2 ]].

  [[ Ref(v) ^ (s) : ty ]] :=
    Var{v} -> s,
    Var{v} |-> d | error $[Cannot resolve
    [v]] @ v,
    d : ty.
```

Fig. 5 An NaBL2 code snippet that declares name binding and typing for EXP

```
strategies

  simplify0 = bottomup(try(simplify0-term))

rules

  simplify0-term: Add(Integer("0"), x) -> x
  simplify0-term: Add(x, Integer("0")) -> x
  simplify0-term: Mul(_, Integer("0")) ->
    Integer("0")
  simplify0-term: Mul(Integer("0"), _) ->
    Integer("0")
```

Fig. 6 A Stratego code snippet that defines transformations on EXP for simplifying expressions

2.5 Transformation with Stratego

Stratego [19, 38] is a transformation language based on term rewriting and programmable rewriting strategies. Rewrite rules specify how a single input term transforms into an output term. These rules can be combined by putting them in sequence (e.g., `r1; r2`), by non-deterministically choosing between them (e.g., `r1 + r2`), or they can be passed to pre- or self-defined AST traversal strategies such as `topdown(r1)` or `bottomup(r1)`.

See Fig. 6 for an example Stratego transformation for EXP. Strategy `simplify0` simplifies expressions that contain zeroes by performing a bottom-up traversal through the AST and trying to apply rule `simplify0-term` on every AST node. The rule `simplify0-term` is defined four times, each for a different type of expression that can be simplified. Each `simplify0-term` rule is tried until the AST node matches with the left-hand side of the rule, after which the transformation is applied. The order in which the rules are tried is chosen non-deterministically during runtime. The `try` rule allows each `simplify0-term` rule to fail, which can happen, for instance, when the AST node is an `Int` term, after which it simply continues with the traversal.

```
strategies

  print-exp = bottomup(print-term)

rules

  print-term: Int(x) -> x
  print-term: Ref(v) -> v
  print-term: Add(x, y) -> $[[x] + [y]]
  print-term: Mul(x, y) -> $[[x] * [y]]
  print-term: Let(v, x, y) -> $[let [v] =
    [x] in [y]]
```

Fig. 7 A Stratego code snippet that defines a printer for EXP

Transformations with Stratego are generally model-to-model, which can be both endogenous (source and target are the same language) and exogenous (source and target are different languages) [39]. Figure 6 is an example of an endogenous model-to-model transformation. It is also possible with Stratego to define model-to-text transformations, since a string is a valid term too. Stratego supports such transformations with *templates*, denoted with `$[..]`. A template defines a string in which variables and transformation rules can be used to create substrings. See Fig. 7 for a transformation that prints an EXP AST. Note that given the syntax definition of EXP, such a pretty printer is generated automatically.

2.6 Editor services with ESV

ESV is a language for defining editor services. An ESV specification can, e.g., customize syntax highlighting coloring and configure editor actions. See Fig. 8 for an ESV snippet that adds an editor action for EXP. This snippet defines a new menu called `Simplifications`, consisting of an action `Simplify zeroes`. This action is mapped to the transformation `editor-simplify0` (definition not explicitly shown), which applies `simplify0` to an EXP specification. These actions can be invoked in the IDE via the menu `Simplifications / Simplify zeroes` whenever an EXP file is in focus.

```
menus
  menu: "Simplifications" (openeditor)
    action: "Simplify zeroes" =
    editor-simplify0
```

Fig. 8 An ESV code snippet that adds an editor action to simplify expressions with zeroes in an EXP specification

```
1  language EXP
2
3  test simplify addition with zero [[
4  3 * x + 0
5  ]] transform "Simplifications / Simplify
      zeroes" to EXP [[
6  3 * x
7  ]]
```

Fig. 9 An SPT code snippet that defines a test for `simplify0`

2.7 Testing with SPT

SPT [40] is a language testing framework for languages implemented in Spoofax. In SPT, test programs can be written and tested for errors and expected outputs. For testing static analysis, one can, e.g., provide an incorrect program where some elements have been marked. Then in the test expectation one can specify at which of these markers an error should occur. For testing transformations, one can provide an input program, an editor action to execute, and an expected output program. Such a test compares the AST that results from the editor action to the AST that results from parsing the expected specification, so the formatting of the provided specifications does not influence the test.

See Fig. 9 for an SPT snippet that defines a test for the `simplify0` transformation. Line 4 defines the input specification, line 5 defines the editor action to apply, and line 6 defines the expected output specification.

3 OIL

We first give an overview of OIL. Afterward, we define a number of features that should be realized by the implementation of OIL in Spoofax.

3.1 History of OIL

OIL, which stands for Open Interaction Language, is a language developed by Van Gool (co-author) to model the behavior of control-software systems. In its early stages, OIL was designed to model the intended communication behavior between a group of components, known as a *protocol*. Later, OIL was adapted to also enable the modeling of individual components. Although OIL is developed at Canon Production Printing, it is not limited to modeling systems within the printing domain. The original syntax of OIL is XML-based, but a more user-friendly DSL variant was created using Spoofax [41]. Though OIL is a textual language, it was designed to allow for an unambiguous visualization, as this is indispensable for communication between engineers.

With the development of OIL also came dedicated tooling. This tooling, implemented in Python, is able to parse and validate OIL specifications. It is a web-based environment in which OIL specifications can be inspected but not edited; editing happens inside a separate IDE, typically Visual Studio. The tooling also supports the visualization of OIL specifications, as well as simulation of traces over this visualization. For OIL component specifications, it can generate executable code, which has been used to implement several complex software components for printers developed at Canon Production Printing. In this web-based tooling, OIL specifications can be pretty printed and editor services such as syntax highlighting and error reporting are available. In the rest of this document, we refer to this implementation of OIL as “the Python implementation”.

3.2 Overview of OIL

OIL is a state machine language that uses variables to store the current state. These variables and their values can be represented by areas, which are connected with transitions that can specify updates of variables, triggered by the occurrence of events. In this section, we give an informal description of the concepts of an OIL component specification and their semantics that are relevant for this paper. For a more in-depth description of OIL and a definition of its formal semantics, see [42].

We use the OIL component specification in Fig. 10 as running example. This OIL specification models a printer that, after a client has registered, can be turned on and off. When it is on, jobs of at most three sheets can be sent to the printer that are immediately processed. The printer also keeps track of its temperature and must be cooled down if it becomes too hot. See Fig. 11 for a visualization of the running example.

Each OIL component specification defines a number of *instance variables* (lines 9–12), which store the state that the modeled component is in. Four types of instance variables are supported: boolean, enum, integer and component instance reference. Enum types can be defined within the specification itself (line 7).

There are three types of areas: *regions*, *states* and *zones* (lines 14–33). A region always refers to an enum variable and contains a number of states. These states each represent a value that the variable of its region can have. A zone has

```
 1  component heat2c
 2  {
 3    import heat2ci
 4    provides heat2ci.server
 5    requires heat2ci.client
 6
 7    enum power {off, on}
 8
 9    var power : power
10    var client : heat2ci.client
11    var tmp : int32 = 20
12    var sheets : int32 = 0
13
14    state init
15    state active
16    {
17      region power [this.power]
18      {
19        state off ['off']
20        state on ['on']
21      }
22
23      zone power_on [this.power == 'on']
24      {
25        region job
26        {
27          state idle
28          state busy
29        }
30      }
31
32      zone heat [this.tmp < 45]
33    }
34
35    concern REGISTRATION
36    {
37      in init on register_client() assign this.client :=
        client go active end
38    }
39
40    concern POWER
41    {
42      in off on turn_on() go on end
43      in on on turn_off() go off end
44    }
45
46    concern JOB
47    {
48      in idle on add_job(nrsheets) if nrsheets > 0 and
        nrsheets <= 3 assign this.sheets := nrsheets go busy
        end
49      in busy if this.sheets == 0 do [silent] job_printed()
        go idle end
50      in busy if this.sheets > 0 at this.client do
        sheet_printed(sheetnr = this.sheets) assign
        this.sheets := this.sheets - 1 go busy end
51    }
52
53    concern HEAT
54    {
55      in heat on turn_on() assign this.tmp := this.tmp + 5
        go heat end
56      in heat if this.tmp > 20 on cool_down() assign
        this.tmp := 20 go heat end
57    }
58  }
```

Fig. 10 The OIL specification for an overheating printer (in the newer DSL notation)

a Boolean expression over variables and is used to restrict behavior.

The change of values for instance variables is triggered by the occurrence of *events*. Each event has an *operation*, which refers to the function being called. This operation may have parameters, which make it possible to transfer data between components. In the context of a component, the *cause* of an event can be either *reactive* or *proactive*. Reactive events are initiated by the environment, whereas proactive events are produced by the component itself, either sent to the environment or kept internally, the latter are also known as *silent* events (Fig. 12).

Operations are declared in separate specifications in a language called IDL, short for Interface Definition Language (very similar to, but not to be confused with Microsoft's IDL[4]). Each IDL specification defines a number of *modules*, which contain *interfaces*, which in turn contain declarations of operations, possibly with parameters. A module may also contain enum type definitions, which can be used to define the type of a parameter. If one wants to refer to operations within an OIL component specification, the IDL modules in which they are defined must be imported (line 3, Fig. 10). Interfaces in the imported module can then be *provided* or *required* by the component (lines 4–5). The operation of a reactive event must be part of a provided interface, and the operation of a proactive event must be part of a required interface.

The occurrence of an event corresponds to the firing of transitions labeled with that event (lines 35–57). Each transition has a source area (`in`), an event label (`on`/`do`), a target state (`go`) and a concern (`concern`), and optionally a guard (`if`), assignments (`assign`), an assertion (`assert`, not in example) and arguments for parameters (line 50, within parentheses).

3.3 Implementation features

The desired implementation of OIL in Spoofax is required to realize a number of features. Though these features may not be complex to implement individually, the realization of the combination of these features can be. Below, we elaborate on each OIL feature (OF).

OF 1 Multiple Syntaxes. OIL and IDL both offer an XML-based and a custom DSL syntax. Both syntaxes should be implemented; the XML syntax for backwards compatibility and because it is easier to parse for external tools, and the DSL syntax for a better user experience. It should be possible to transform a specification in one syntax into the other and all transformations to other targets should be available for both syntaxes.

OF 2 Desugaring. OIL specifications allow some syntactic sugar, mainly in the form of leaving OIL concepts implicit, which reduces how much a user needs to write. For instance, in the running example, for region `job` (line 25) the variable reference is left implicit and for states `init`, `active`, `idle` and `busy` (lines 14, 15, 27 and 28) the values are left implicit,

[4] https://learn.microsoft.com/en-us/windows/win32/midl/midl-start-page.

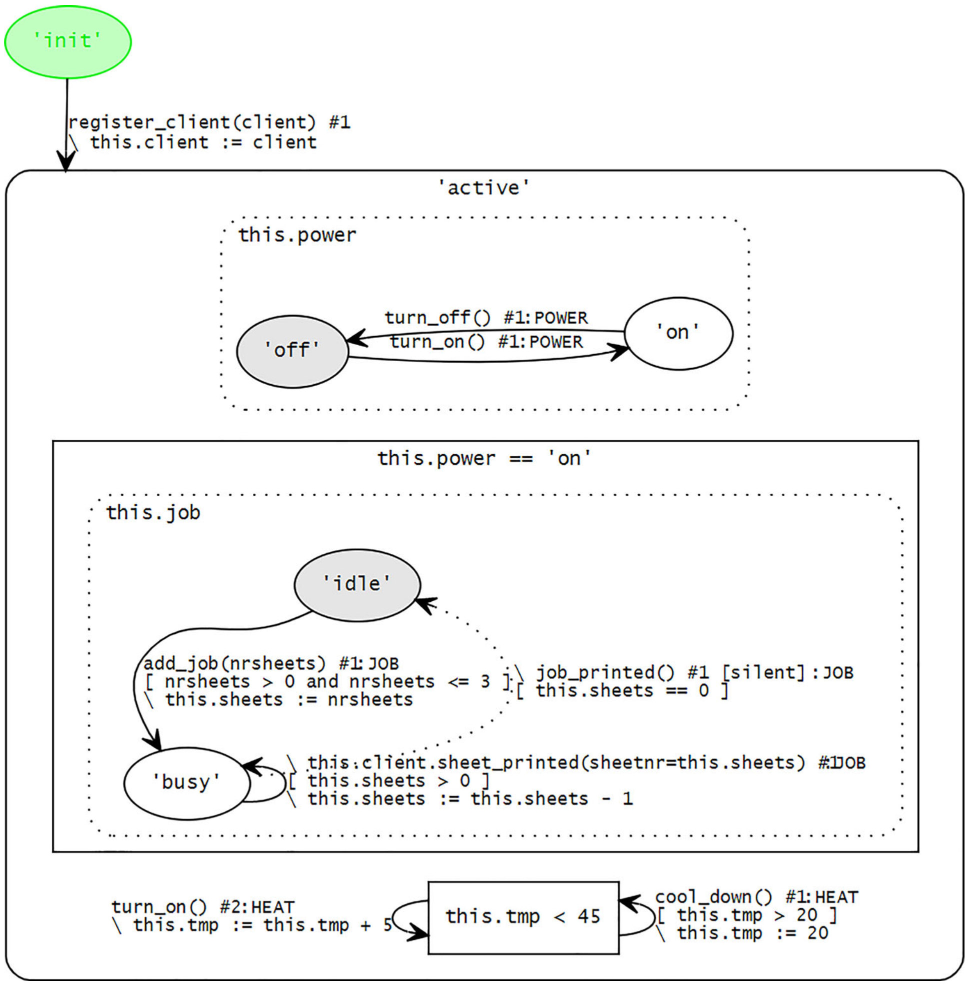

Fig. 11 A visualisation of the example OIL specification of Fig. 10. States that are filled with a color correspond to the initial state.

as well as corresponding enum types definitions. Also, the region for states `init` and `active` is left out. See Fig. 13 for how the first half of the running example would look like after desugaring. The implementation should be able to automatically desugar this and make the implicit information explicit.

OF 3 Input Correctness. Not every specification is a correct OIL specification in terms of syntax or static semantics. Checking the static semantics of an OIL specification involves three types of analysis: structural checks, name resolution and type checking. The implementation should be able to check whether a specification meets all syntax and static semantics requirements and report useful errors to the user if it does not.

OF 4 Language Interaction. IDL is a standalone language that can be used for other purposes than the context of OIL. OIL on the other hand should depend on IDL, both syntactically and semantically. Syntactically, because both languages use expressions and we want to minimize duplicate grammar definitions. Semantically, because module, interface, operation and parameter names in OIL specifications should refer to declarations in IDL specifications. The implementation should reflect this: IDL should be implemented separately

```
module heat2ci
{
  interface server
  {
    register_client(client: heat2ci.client)
    turn_on()
    turn_off()
    add_job(nrsheets: int32)
    cool_down()
  }

  interface client
  {
    sheet_printed(sheetnr: int32)
  }
}
```

Fig. 12 The IDL specification on which the OIL specification of Fig. 10 depends

```
component heat2c
{
  import heat2ci
  provides heat2ci.server
  requires heat2ci.client

  enum power {off, on}
  enum t_root_region {init, active}
  enum t_job {idle, busy}

  var power : power = 'off'
  var client : heat2ci.client
  var tmp : int32 = 20
  var sheets : int32 = 0
  var v_root_region : t_root_region = 'init'
  var v_job : t_job = 'idle'

  region root_region [this.v_root_region]
  {
    state init ['init']
    state active ['active']
    {
      region power [this.power]
      {
        state off ['off']
        state on ['on']
      }

      zone power_on [this.power == 'on']
      {
        region job [this.v_job]
        {
          state idle ['idle']
          state busy ['busy']
        }
      }

      zone heat [this.tmp < 45]
    }
  }
  ...
}
```

Fig. 13 The first half of the OIL specification of Fig. 10 after desugaring

and the implementation of OIL should depend on the implementation of IDL. This involves several forms of language composition [43] and language modularity [44, Sec. 4.6].

OF 5 Multiple Targets. To represent the dynamic semantics of an OIL specification, it should be possible to translate OIL into other languages for which such semantics exists. For the formal verification of an OIL specification, the implementation should support a translation to mCRL2 [45]. For the execution of an OIL specification, the implementation should support a translation to GPL code.

4 Case study context and method

In this section, we first describe the context of our case study. Next, we elaborate on our method for investigating the research question. Lastly, we describe the setup of our case study.

4.1 Context

Our evaluation focuses on two implementations of OIL, the Python implementation and the Spoofax implementation. The Python implementation was initially developed around 2011 by the third author and is still maintained by the third author to this day. The first author also worked on the Python implementation for a few months in 2016 as part of an internship within Canon Production Printing. The second author initiated the Spoofax implementation in 2018. A few months later, the first author also joined on the Spoofax implementation and both first and second authors have been maintaining this implementation ever since. During this time, the third author was involved in the design decisions for the Spoofax implementation and some master students have contributed as well [46–48].

Before the Spoofax implementation was created, the second author was already familiar with language development in Spoofax. The second author has also been a contributor to Spoofax since before this study. The first author had limited experience in language development and no experience with Spoofax, but had some previous experience on rewriting and formal semantics. During the development of the Spoofax implementation, the developers had a close connection to the Spoofax development team for any questions and advice. All involved master students had no experience with Spoofax before they joined.

The third author is the creator of OIL, inspired by his prior research in the field of the specification of behavior [49]. The first author got experience with OIL during the internship, in which the goal was to understand and formalize the semantics of OIL by means of a(n) (initial) translation to mCRL2, on which the current translation to mCRL2 in Spoofax is based [50]. Prior to that, the first author had experience with behavior specification languages, specifically mCRL2. The second author had little experience with behavior specification languages before the development of OIL. All involved master students had no experience with OIL before they joined, but most had some experience with behavior specification languages.

4.2 Research method

Productivity is about the amount of effort needed to implement some functionality. As proxy for effort we use code volume, as it is the only information available to us in this study that relates to effort. To represent functionality, we collect software artifacts relevant to language engineering that an implementation produces, such as parsers or transformations.

We compare the implementation of OIL in Spoofax with the implementation of OIL in Python. We do this by first gathering all artifacts relevant to language engineering. Then, for each artifact implemented in both implementations with similar functionality, we measure the code volume that is used to implement it and compare the measured code volume between the two implementations. In case parts of the code volume are reused for multiple artifacts or other projects, we measure it separately. Any dissimilarities between implementations of a language engineering artifact are discussed.

We use the *Source Lines of Code* [51] (SLOC) metric for measuring code volume, which excludes blank lines, comments, and lines only containing brackets from counting. In particular, we use the *Physical* SLOC metric [51], which considers each non-excluded line as a single line. This is in contrast to the *Logical* SLOC metric [51], which counts executable statements of which there could be multiple on a single physical line. Since the Spoofax meta-DSLs are declarative and the code written in these meta-DSLs do not necessarily correspond directly to statements or units of execution, we cannot measure Logical SLOC for both implementations. In the rest of this paper, we use "SLOC" to refer to Physical SLOC.

We are aware that using code volume, quantified using a variant of the *Lines of Code* metric, is controversial and comes with advantages and disadvantages [51–54]. However, an important motivation for using code volume per artifact as proxy for productivity is that it is an objective and repeatable measure and that it is applicable to both the Python and the Spoofax implementation.

Comparing code volume of the two implementations is only sensible when the volumes correspond to code that implements the same functionality. Since the two implementations do not always implement the exact same functionality, we first identify commonalities and differences before measuring volume. Then, in each implementation's code volume measurement, we subtract lines for features or language constructs that are not in the other implementation to end up with a comparison of code that implements the same functionality. We do these measurements separately for artifacts in both implementations. We analyze where differences in code volumes originate from and to what extent parts of the implementations are reusable. We consider code to be reusable if it is generic enough such that it can be reused for other purposes, such as other language implementation, or for purposes outside of OIL altogether.

Depending on the artifact that is being compared, the relevant code consists of whole files or parts of files. When measuring code volume of whole files, we use the cloc tool[5] for the measurements, which counts SLOC and has builtin support for Python. To use this tool on measuring code written in Spoofax meta-DSLs, we manually add language definitions to cloc such that the tool can properly detect which lines need to be excluded from counting, such as lines with a single square bracket. When measuring code volume in parts of files, we count lines of code by hand. With the code measurements that we give, we also go into more detail on how we came to these measurements.

4.3 Setup

We answer the research question for the implementation aspects of language engineering separately. These are concrete syntax, abstract syntax, static semantics, dynamic semantics and design environment [55]. Since the design environment, which is about tool support for the language, is claimed to be (mostly) automatically derived by Spoofax, we do not consider this aspect separately, but as part of the other four aspects instead. Since the Python implementation does not have a dedicated text editor, this will only concern editor services such as syntax highlighting. Thus, we consider the following four aspects:

- **Concrete syntax** (Sect. 5): the textual representation of a language.
- **Abstract syntax** (Sect. 6): the internal representation of a language, including desugaring transformations defined on it.
- **Static semantics** (Sect. 7): the validity of specifications in a language.
- **Dynamic semantics** (Sect. 8): the execution of specifications in a language.

In each of these four sections, we first highlight parts of the implementation that are relevant to the aspect. Afterward, we evaluate Spoofax by answering the research question in the context of the aspect on a number of parts of the implementation, which we call *evaluation points*. For each evaluation point, we structure the evaluation in the following parts:

- **Question**: what do we want to evaluate?
- **Method**: how are we going to evaluate this?
- **Results**: what is the information from the implementation(s) that is relevant for this evaluation point?
- **Analysis**: what does this information mean and how does it answer our question?
- **Conclusion**: what does the analysis give as answer to the question?
- **Discussion**: What else is relevant for this evaluation point?

We combine and summarize the findings of the evaluation points in Sect. 9.1. Since code volume per artifact does not exactly correspond to productivity [51–54], our measurements are not directly representative for the research question. Therefore, in Sect. 9.2, we discuss the threats to

[5] https://github.com/AlDanial/cloc.

the validity of our findings and how we have tried to counter them.

While our evaluation is mostly based on drawing conclusions from a quantitative analysis, we also make various observations on qualitative aspects. In Sect. 10, we discuss those observations on qualitative aspects. We discuss the strengths and weaknesses of Spoofax that we experienced and list the lessons we learned. We also propose an agenda of future work for Spoofax and discuss how some of the limitations that we encounter are already fixed in the next version of Spoofax.

5 Concrete syntax

The Spoofax implementation of OIL comprises multiple syntactical (sub)languages for which the grammar is defined with SDF3. It supports the original XML syntax of OIL and IDL as supported by the Python implementation, as well as a newly designed custom syntax, resulting in a total of four input languages and realizing OF1 (Multiple Syntaxes). These input languages share common grammar rules for expressions, which touches on OF4 (Language Interaction).

In this section, we describe the design of the existing and new syntaxes in Spoofax, their modular implementation, and the reuse of shared expression grammar rules and its implications on disambiguation. Also, we describe concrete syntax in the Python implementation and indicate how it differs from the Spoofax implementation. These descriptions form the sources of information for the evaluation that follows, where we answer the research question on productivity for the concrete syntax aspect of OIL's implementation in Spoofax.

5.1 From XML to custom syntax

Implementing a language with concrete syntax in Spoofax requires a grammar written in SDF3. The grammar specific for the OILXML subset of XML consists of a grammar rule for each XML element, with, if applicable, a list of specific attributes of the element and, if applicable, a list of child elements. Figure 14 shows an excerpt of the SDF3 grammar of OILXML for the transition concept (see Sect. 3.2). Figure 16a contains an example transition in OILXML, and Fig. 16c contains the corresponding abstract syntax, expressed in ATerm (see Sect. 2.2).

The original Python implementation uses XML as its syntax because of two reasons. First, other projects at Canon Production Printing already used XML and thus engineers are familiar with it. Second, for XML an off-the-shelf parser could easily be used. However, writing XML specifications by hand is not user friendly [56, p. 101]. Although a custom syntax was desired already in the Python implementation to improve usability, implementing it was considered too much work. With Spoofax's language-oriented programming view [57], re-implementing OIL in Spoofax made it much more feasible to design a custom syntax for OIL, dubbed "OILDSL".

```
context-free syntax

  Transition.XMLTransitionSimple = [
    <transition[{TransitionAttr ""}+]/>
]

  Transition.XMLTransitionComplex = [
    <transition[{TransitionAttr ""}*]>
        [TransitionSelf?]
        [{TransitionParameter "\n"}*]
        [TransitionReturn?]
        [TransitionGuard?]
        [TransitionAssignments?]
        [TransitionAssert?]
    </transition>
]

  TransitionAttr.XMLMessageCauseAttr     =
    [cause="[Cause]"]
  TransitionAttr.XMLMessageOperationAttr =
    [operation="[ID]"]
  TransitionAttr.XMLSourceAttr           =
    [source="[AreaReference]"]
  TransitionAttr.XMLTargetAttr           =
    [target="[AreaReference]"]
```

Fig. 14 An SDF3 code snippet of the grammar of transitions in OILXML

The most prominent problem of XML syntax is the syntactic noise of XML elements and attributes. Still, the same high-level structure of OIL's concepts from OILXML was used as a basis for designing the new syntax. By doing so, the abstract syntax of both syntaxes is similar, which eases forward and backward migration between both syntaxes. Without the noise of XML elements in OILDSL, the syntax for transitions becomes simpler; XML open and closing elements are replaced with simple keywords and brackets. Figure 15 shows an excerpt of the SDF3 grammar for transitions in OILDSL. Figure 16b shows the concrete syntax in OILDSL that corresponds to the OILXML variant in Fig. 16a, and Fig. 16d depicts the corresponding AST.

5.2 Composed grammars and disambiguation

The four input languages (IDLXML, IDLDSL, OILXML and OILDSL) share parts of their grammars, especially in the context of expressions: the expression grammar of OIL extends the expression grammar of IDL, and the XML and DSL expression grammars only differ in a few operators. To prevent the definition of duplicate grammar rules across the

```
context-free syntax // Transitions

  Transition.DSLTransition =
    [[{TransitionElement " "}+] end]

  TransitionElement.DSLMessage =
    [[MessageCause] [MessageEventType?]
    [MessageOperation?]]

  MessageCause.DSLMessageCause = Cause
  Cause.DSLReactive  = [on]
  Cause.DSLProactive = [do]

  TransitionElement.DSLSource =
    [in [AreaReference]]
  TransitionElement.DSLTarget =
    [go [AreaReference]]
```

Fig. 15 An SDF3 code snippet of the grammar of transitions in OILDSL

```
<transition cause="reactive"
    operation="turn_on"
  source="off" target="on"/>
```

(a) Example transition in OILXML.

```
in off on turn_on() go on end
```

(b) The transition of (a) translated to OILDSL.

```
XMLTransitionSimple(
  [ XMLMessageCauseAttr(XMLReactive())
  , XMLMessageOperationAttr("turn_on")
  , XMLSourceAttr(AreaReference(["off"]))
  , XMLTargetAttr(AreaReference(["on"]))
  ]
)
```

(c) The OILXML AST that corresponds to (a).

```
DSLTransition(
  [ DSLSource(AreaReference(["off"]))
  , DSLMessage(
      DSLMessageCause(DSLReactive())
      , None()
      , Some(DSLMessageOperation("turn_on",
    [], None()))
    )
  , DSLTarget(AreaReference(["on"]))
  ]
)
```

(d) The OILDSL AST that corresponds to (b).

Fig. 16 An example transition in OILXML and OILDSL with the corresponding ASTs

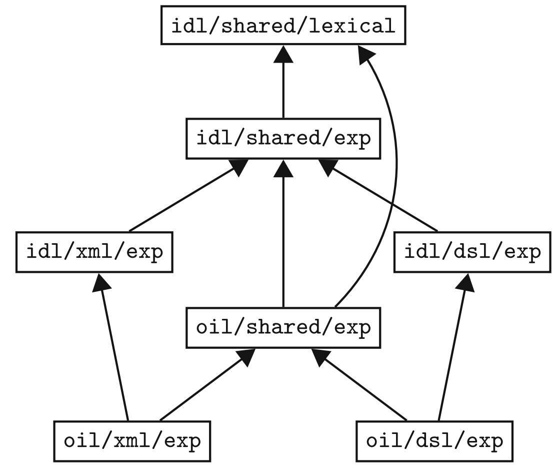

Fig. 17 A (simplified) import graph of IDL's and OIL's expression grammars, that depicts how modules are reused. Node labels correspond the SDF3 module names. Arrows mean "imports"

SDF3 grammar definitions of the input languages, the grammar definitions are split up into several reusable modules. See Fig. 17 for all SDF3 modules that define grammar rules for expressions. Modules `idl/xml/exp`, `idl/dsl/exp`, `oil/xml/exp` and `oil/dsl/exp` define the expression grammar for IDLXML, IDLDSL, OILXML and OILDSL respectively. Shared grammar is defined in separate modules and reused for multiple grammars by means of `import` statements. As an example, Fig. 18 shows the composition of the IDLXML expression grammar with the shared OIL expression grammar (module `oil/shared/exp`) to obtain the OILDSL expression grammar in SDF3. The XML-specific grammar rules are defined once for IDLXML in module `idl/xml/exp` and are reused for OILXML by importing them in module `oil/xml/exp`. Also, the OIL-specific grammar rules, used for both OILDSL and OILXML, are imported from module `oil/shared/exp`. Together, both imported modules form the expression grammar of OILXML.

Module `idl/xml/exp` has emerging ambiguities between the XML-specific operators and other operators. Syntactical ambiguity is when a single input program can be parsed to multiple different trees using the same grammar. For example, `1 + 2 &gt;= 3` could be parsed in two ways: `(1 + 2) &gt;= 3` and `1 + (2 &gt;= 3)`. With priorities in SDF3, one can define the relative precedence of expression operators. Since priorities are transitive, only priorities with respect to direct neighbors in the priority order (operators `Plus`/`Minus` and `Eq`/`Neq`) are required for the new operators. Because `Plus` gets a higher priority than `XMLGeq`, the previous example will be parsed as `(1 + 2) &gt;= 3`. In module `oil/xml/exp`, no additional disambiguation is required, as no more new combinations of expression operators arise and no other syntactical ambiguities emerge.

```
module idl/xml/exp

imports idl/shared/exp

context-free syntax
  Exp.XMLLt   = [[Exp] &lt; [Exp]]  {left}
  Exp.XMLLeq  = [[Exp] &lt;= [Exp]] {left}
  Exp.XMLGt   = [[Exp] &gt; [Exp]]  {left}
  Exp.XMLGeq  = [[Exp] &gt;= [Exp]] {left}

context-free priorities
  { left: Exp.Plus Exp.Minus } >
  { left: Exp.XMLLt Exp.XMLLeq Exp.XMLGt
    Exp.XMLGeq } >
  { left: Exp.Eq  Exp.Neq }
```

```
module oil/shared/exp

imports idl/shared/Lexical
imports idl/shared/exp

context-free syntax
  // OIL-specific expressions
  Exp.Reference = ID
  Exp.Old       = [[Exp]']

context-free priorities
  Exp.Has >
  Exp.Old >
  { left: Exp.Not Exp.Length }
```

```
module oil/xml/exp

imports idl/xml/exp
imports oil/shared/exp
```

Fig. 18 SDF3 modules that define the expressions syntax for OILXML (simplified). The `Exp.Reference` constructor defines a variable reference using the lexical `ID` sort. The `Exp.Old` constructor defines a suffix operator for referencing old values of a variable, which can be used in assertions

5.3 The Python implementation

The Python implementation only supports IDLXML and OILXML. The parser for these languages consists of two stages. First, the XML structure is parsed using the Minidom library.[6] Second, the expressions inside XML elements are parsed using the Pyparsing library.[7] For both stages, the implementation uses a custom layer on top of the libraries to support grammar specification and parser implementation. For example, the Python implementation uses a metamodel expressed in a custom XML subset that defines the restrictions of OILXML with respect to generic XML.

Figure 19 shows an excerpt of this metamodel. Lines 2–4 define a regular expression for identifiers that can be referred to in other parts (see, e.g., line 10). Lines 6–33 define expressions, in which the order of levels of operators indicates the precedence between operators. Lines 35–47 define the transition concept. A Python script parses this metamodel and checks for an input OIL program, parsed using Minidom, whether it conforms to the metamodel. Helper functions on top of Pyparsing ease the definition of expression grammar rules, e.g., by automatically allowing whitespace around operators. The levels of operators are used to define precedence in Pyparsing. The custom layer is not specific to OIL and can be reused for other XML languages with embedded expressions.

In Sect. 3.1, we have described the Python implementation's environment for viewing and editing OIL specifications. Several editor services related to viewing and editing concrete syntax similar to those in regular IDEs are available, which we detail below. For viewing OIL specifications, the Python implementation supports pretty printing and origin tracking (for error reporting, see Sect. 2.3), which are implemented manually. Both pretty printing and origin tracking are implemented in Python based on the data structures that result from the parsers.

For editing OIL specifications, the Python implementation has limited custom support. The web-based Python implementation does not include an editor, so external tools are used instead for editing OIL specifications, typically the Visual Studio IDE. To support the editing of OIL specifications, an XSD (XML Schema Definition Language[8]) file is generated from the metamodel using a handwritten script. XSD schemas define a subset of the XML language. A generic IDE plugin for XML uses this XSD file to provide syntactic code completion and error recovery. Although XSD could also be used for realizing a parser, it was only used for realizing limited editor support because XSD was found not to be expressive enough to cover all syntactical aspects of OIL.

5.4 Evaluation

To evaluate the productivity of implementing concrete syntax, we look at a single evaluation point: the complete implementation of concrete syntax in OIL's implementations in Spoofax and in Python. We consider seven concrete syntax artifacts: the grammar, the parser, and the editor services pretty printing, origin tracking, syntax highlighting, error recovery and syntactic code completion.

Question. Does it cost less code volume to implement the artifacts for OIL's concrete syntax in Spoofax compared to Python?

6 https://docs.python.org/3/library/xml.dom.minidom.html.
7 https://pypi.org/project/pyparsing/.
8 https://www.w3.org/XML/Schema.

```
<metamodel name="oil">
    <regexp name="identifier" pattern="[_A-Za-z][_0-9A-Za-z]*">
        <documentation>A standard programming identifier.</documentation>
    </regexp>
    ...
    <parexp name="expression">
        <level>
            <operator symbol="number" pattern="0[1-9][0-9]*"/>
            <operator symbol="new" pattern="new {qualified_identifier}"/>
            <operator symbol="identifier" pattern="{identifier}"/>
            ...
        </level>
        ...
        <level>
            <operator symbol="_+_"/>
            <operator symbol="_-_"/>
        </level>
        <level>
            <operator symbol="_==_"/>
            <operator symbol="_!=_"/>
            <operator symbol="_&lt;=_"/>
            <operator symbol="_&gt;=_"/>
            <operator symbol="_&lt;_"/>
            <operator symbol="_&gt;_"/>
        </level>
        <level>
            <operator symbol="_and_"/>
        </level>
        <level>
            <operator symbol="_or_"/>
        </level>
        ...
    </parexp>
    ...
    <entity name="transition">
        <documentation>A transition specifies a rule that indicates for a certain action when
            and how variables change.</documentation>
        <generalization entity="actionable"/>
        <generalization entity="compositional"/>
        <generalization entity="concernable"/>
        <child entity="self" lower="0" upper="1"/>
        <child entity="argument" lower="0" upper="inf"/>
        <child entity="result" lower="0" upper="1"/>
        <child entity="guard" lower="0" upper="1"/>
        <child entity="assignment" lower="0" upper="inf"/>
        <child entity="assertion" lower="0" upper="1"/>
    </entity>
    ...
</metamodel>
```

Fig. 19 An excerpt of the Python implementation's metamodel for OIL, expressed in a custom chosen subset of XML. OIL specifications are checked to conform with this metamodel using a handwritten but generic Python script

Method. First we identify for each of the seven artifacts related to concrete syntax to what extent they are realized in the SDF3 and Python implementations of OIL. Then, per artifact, we measure and compare the code volume (in terms of SLOC [51]) related to the artifact in each implementation.

For the sake of comparability, we want to compare the lines of code of the implementations where they implement the exact same syntax. The Python implementation only contains OILXML and not OILDSL. Thus, in the Spoofax implementation we consider the grammar except those parts specific to OILDSL, i.e., we consider the OILXML-specific parts and the parts shared between OILXML and OILDSL. Since the syntactic languages of both implementations are not exactly the same, we subtract lines from our measurements that concern syntactical elements not present in the other implementation. In the Python implementation's metamodel, we manually exclude tags that are specific for documentation from the counting, as we consider them as comments in the definition of SLOC. Since XML and expression parsing are separately implemented in Python, we measure and analyze those separately.

Both the Python and the Spoofax implementation of OIL could be seen as consisting of OIL-specific code and more generic, reusable code. We consider code to be reusable if it is generic enough such that it can be reused in the implementation of a language other than OIL. In the Spoofax implementation, we reuse code from the standard library of Spoofax, and we do not include it or the implementation of the language workbench itself in the measurements. In the Python implementation, both OIL-specific and reusable code are implemented manually, which we therefore both measure. We count OIL-specific code separately from reusable code. For the reusable code, we analyze to what extent it can be reused.

Results. Table 1 gives an overview of which syntax artifacts are available in each implementation and states the SLOC per artifact. The Spoofax syntax implementation of OILXML comprises 365 SLOC of SDF3 grammar, and the Python implementation comprises 1260 SLOC. Spoofax realizes all artifacts in full from this grammar: a parser, a pretty printer, origin tracking, and editor services such as syntax highlighting, error recovery, and code completion. The Python implementation contains a parser and origin tracking for the full language. Other artifacts are only implemented for XML and not for the expressions inside XML: pretty printing, syntax highlighting, error recovery, and code completion. To ensure that our comparison is on two implementations that cover the same syntactic language, in the Spoofax source we have withheld the syntactical elements that are not in Python from counting (31 out of 396 SLOC deducted from original source; 7.8%) and in the Python source we have withheld elements that are not in the Spoofax implementation from counting (46 out of 1306 SLOC deducted from original source; 3.5%). Of the Python implementation, only the grammar (202 out of 1260 SLOC) is specific to OIL, which relates to the metamodel.

Analysis. We analyze our results by first comparing the total code volumes of both implementations and then compare per syntax implementation artifact.

The results show that the syntax implementation artifacts of OIL are realized in the Spoofax implementation with a factor of 0.29 SLOC compared to the Python implementation. All 365 Spoofax SLOC are OIL-specific. In the Python implementation, only 202 SLOC is specific to OIL, namely for defining the metamodel; the rest is reusable for XML-based languages with embedded expressions. The Spoofax implementation realizes all syntax artifacts for the full language, whereas the Python implementation only realizes the parser and origin tracking for the full language; the other artifacts produced by the Python implementation do not provide support for expressions. The Spoofax implementation consists of SDF3 only, i.e., the grammar, from which all other six artifacts are derived. In the Python implementation, most code is attributed to realizing the parser. For the other artifacts, additional code was needed (90 + 205 + 121 = 416 SLOC). In the Python implementation, syntax highlighting for XML was the only editor service that was available without manual implementation by using an existing XML plugin in Visual Studio.

The Python implementation uses a custom layer on top of existing libraries for XML parsing (Minidom) and expressions parsing (Pyparsing). First, the Python implementation uses a metamodel (202 SLOC) and a script that checks whether OIL models conform to the metamodel (344 SLOC). Second, the Python implementation adds helpers on top of Pyparsing that prevent repeating low-level grammar patterns (298 SLOC). Although the custom layer for expressions is reusable, it is more restrictive than SDF3 as, e.g., it only supports left associativity. The SDF3 implementation did not use code in addition to the grammar, which is the main reason why the Spoofax implementation contains fewer SLOC.

In Spoofax, the OILXML grammar is defined with a total of 21 SDF3 modules. Out of the 365 SLOC used to define these modules, about 28% exists purely to compose these modules. This consists almost only of module name definitions and import statements. For some grammar modules, such as those that define the expression grammar, the split up into smaller modules is beneficial, because it enables reuse of grammar rules for the other input languages as described in Sect. 5.2. A third of the grammar modules, however, are only used once. These modules could have been merged with the modules that use them instead, which would have saved a few SLOC. We see this difference as negligible, as this only saves on import statements, which do not directly define the grammar of the language. In Python, the grammar is defined in a single metamodel, so no SLOC is used for composition.

Table 1 Code volume (in SLOC) for the Spoofax and Python implementations of OILXML's concrete syntax, counted per implementation artifact

Syntax impl. artifact	Spoofax	Python
Grammar (excl. expressions)*	● 274	● 161
Grammar (expressions)*	● 91	● 41
Parser generator (excl. expressions)	● 0	● 344
Parser generator (expressions)	● 0	● 298
Pretty printing	● 0	◐ 90
Origin tracking	● 0	● 205
Syntax highlighting	● 0	◐ 0
Error recovery Syntactic code completion	● 0	◐ 121
Total (OIL-specific)*	365	202
Total (All)	365	1260

*Indicates OIL-specific artifacts. ●, implemented; ◐, partially implemented (only for XML, not for expressions)

Pretty printing is implemented manually in the Python implementation (90 SLOC). This is a generic XML pretty printer, not specific to OILXML. For expressions, it simply copies the textual representations of expressions to output programs. In the Spoofax implementation, pretty printing is automatically derived based on the formatting of the grammar rules in SDF3. For example, lines 7–16 of Fig. 14 define the pretty printing of transitions to be spread over multiple lines and with indented child elements. These templates increase the number of SLOC used for defining the grammar rule, as without formatting the complete rule could be at a single line.

Origin tracking (see Sect. 2.3) is generated automatically from the SDF3 grammar. In the Python implementation, origin tracking is only provided without requiring additional implementation by the Pyparsing library used for parsing expressions. For parsing XML, a manual implementation was needed to realize origin tracking (196 SLOC). The origins returned by the Pyparsing library used for expressions are relative. Absolute source locations for expressions are calculated by adding the relative offsets of expressions to the parent XML tag content's source location (9 SLOC, reported under "Origin tracking" in Table 1).

Syntax highlighting is the only editor service in Python that is realized without additional effort, using Visual Studio's default XML plugin. Note, however, that this only supports syntax highlighting for the XML part of OIL. For the expressions inside XML tags, it does not support syntax highlighting. The Spoofax implementation does support syntax highlighting for the complete language.

The Python implementation realizes error recovery and code completion for the XML part of the language by generating an XSD schema from the metamodel. The script that generates the XSD schema (121 SLOC) is reusable, not specific to OIL. Given the XSD schema, the editor's default XML plugin provides error reporting on invalid XML tags and it provides code completion. The Spoofax implementation supports error recovery and code completion for the complete language without additional implementation, by automatically generating the editor services based on the SDF3 grammar.

Conclusion. The Python implementation uses less OIL-specific code to implement OIL's concrete syntax (metamodel of 202 SLOC) than Spoofax (SDF3 grammar of 365 SLOC). However, whereas Spoofax does not require code in addition to the grammar to fully implement a range of editor services, the Python implementation requires additional code for implementing a parser and other editor services. This is in spite of the Python implementation making use of existing libraries for XML parsing and expression parsing and the Python implementation realizing some editor services not for the full OIL language. The code in the Python implementation, apart from the metamodel, is reusable in the sense that it is not specific to OIL. The reusable parts of the Python implementation can be reused for implementing other languages with XML syntax with a restricted form of embedded expressions.

Comparing the implementations, including the reusable parts, the Spoofax implementation realizes all editor services for the full OIL language using less than one third of SLOC. Therefore, the results show that it costs less code volume to implement the artifacts for OIL's concrete syntax compared to Python. This is especially the case when, in addition to only a parser, editor services are required for interactive use in IDEs.

Discussion. The Python implementation uses, on top of the existing parsing libraries, a custom layer to support the implementation of OILXML's syntax. This custom layer enables the use of the metamodel and prevents repeating low-level grammar patterns. Alternatively, OILXML's syntax could also have been implemented directly using the existing libraries. On the one hand, the custom layer is reusable and costs extra code. On the other hand, the custom layer saves code by preventing the need to repeat low-level implementation patterns. Although we cannot make a comparison with a Python implementation that does not include the custom layer, we expect that such an implementation could be smaller than the current Python implementation. An implementation without reusable parts could be more specific to OIL and therefore possibly smaller. In the Spoofax implementation, SDF3 was sufficient to implement OILXML and OILDSL.

Implementing the parser for OILXML in Python takes about twice the SLOC as using SDF3. When editor support is not relevant, one could argue whether the factor two fewer SLOC is reason enough to start using a language workbench. However, we expect that the factor is relatively low because XML syntax is used. By using XML, the existing Minidom library could be used, but this also imposes the restriction of only supporting subsets of XML. For custom syntax, e.g., for OILDSL, we expect that a Python implementation using Pyparsing would cost relatively more SLOC than in SDF3, as a complete grammar needs to be implemented in more detail. The use of another parser generator library in Python could bring this closer to SDF3. In Spoofax, the code volume specific to the OILDSL grammar is actually smaller than the code volume specific to the OILXML grammar due to OILDSL being more concise: the Spoofax implementation contains 219 OILXML-specific SDF3 SLOC and 168 OILDSL-specific SDF3 SLOC.

The grammar of OILXML imposes a few restrictions on the order of attributes or child elements of an XML element. This was originally done so that the structure of the derived AST can be made slightly simpler, resulting in slightly simpler transformations from this AST. To lift these restrictions, we believe that about 10 SDF3 SLOC would be necessary. The Python implementation does not have such restrictions.

Part of the Python implementation's editor services, e.g., origin tracking, are implemented only for the web-based tooling which only statically displays specifications and errors and does not support editing specifications. Therefore, the editor services are only used in a static way, in contrast to the interactive way in IDEs. We expect that extending the editor services in the Python implementation to have support for interactive use would cost more code.

6 Abstract syntax

The abstract syntax of a language defines how a language is represented internally. For textual languages, such as OIL, this is done by means of AST schemas. Given an SDF3 grammar definition, a corresponding AST schema is generated automatically. To structure the Spoofax implementation of OIL, additional intermediate representations have been defined, which we discuss here. We also describe the transformation architecture that is shaped around these intermediate representations, specifically focusing on desugaring transformations and on the resilient-staging framework that serves as the basis of this architecture, which realize OF2 (Desugaring). Afterward, we discuss how OIL is internally represented in the Python implementation. Lastly, we evaluate Spoofax on productivity in the context of abstract syntax.

6.1 Intermediate representations

In addition to the AST schemas that are automatically generated by SDF3 for OILDSL and OILXML, three intermediate representations (IRs) are defined for OIL:

- **Normalized IR** A representation that acts as a middle ground between OILDSL and OILXML while still containing as many syntactic details from both languages as possible.
- **Desugared IR** A simplified representation where syntactical details are removed and implicit details are made explicit to enable concise specification of static semantics.
- **Semantic IR** A representation that restructures an OIL specification to ease the implementation of dynamic semantics (code generation) of OIL.

The use of IRs provides separation of concerns: each IR is related to a different language implementation aspect. See Fig. 20 for how the IRs fit in the transformation architecture. Transformations are defined between the normalized IR and both the OILXML and OILDSL AST schemas in both directions so that one can easily switch between OILXML and OILDSL [41]. Any transformations that follow are independent of the concrete syntax used.

To illustrate some differences between IRs, we use the same transition as the example in Fig. 16. See Fig. 21 for this transition in the normalized, desugared and semantic IR. One notable change from OILDSL and OILXML to the normalized IR is that the transition term in the normalized IR now has fixed subterms instead of a list of terms, which was done to make it easier to define transformations on it. When moving to the desugared IR, some optionality is removed by removing `Some` wrapper terms and by replacing `None` terms with information made explicit, such as `Call`. When moving to the semantic IR, transitions are now grouped per event. This is useful for code generation, as an OIL specification is executed by means of sending or receiving events. Also, the source and target are used to define the transition pre- and postconditions (with `ConditionReference`) and the transition update (with `UpdateReference`).

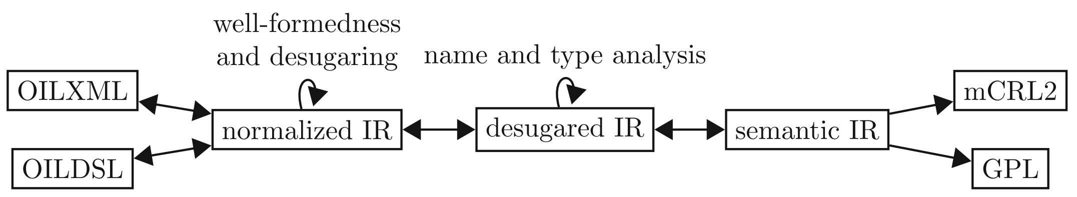

Fig. 20 An overview of the transformation architecture of the implementation of OIL in Spoofax. Boxes correspond to AST schemas and arrows correspond to transformations

```
Transition(
  Some(
    Message(
      Some(MessageCause(Reactive()))
    , None()
    , Some(MessageMethod("turn_on", []))
    )
  )
, Some(Source(AreaReference(["off"])))
, Some(Target(AreaReference(["on"])))
, ...
)
```

```
DESTransition(
  DESMessage(
    Some(Reactive())
  , Call()
  , DESMessageMethod("turn_on", [])
  )
, "off"
, "on"
, ...
)
```

```
SEMEvent(
  "heat2ci_server_turn_on_call_reactive"
, Reactive()
, Call()
, "heat2ci"
, "server"
, SEMMethod("turn_on", [])
, [ SEMTransition(
      ...
    , ConditionReference("off")
    , [UpdateReference("on")]
    , ConditionReference("on")
    )
  , ...
  ]
)
```

Fig. 21 The transition of Fig. 16 in the normalized, desugared and semantic IR respectively

```
1 oil-auto-value =
      topdown(try(oil-auto-value-term))
2
3 oil-auto-value-term:
4   State(name, None(), supers, areas) ->
5   State(name, <oil-auto-value-new> name,
      supers, areas)
6
7 oil-auto-value-new = newname ;
      !Some(StateValue(EnumLiteral(<id>)))
```

Fig. 22 The Stratego implementation of the *auto-value* desugaring transformation

6.2 Desugaring

Before the normalized IR can be transformed to the desugared IR, a number of desugaring transformations are applied first. There are a total of 14 desugaring transformations defined which all use the normalized IR both as input and output. Most of the desugaring transformations are *explications*, which make implicit information explicit. These are necessary to be able to remove the optionality of terms when transforming to the desugared IR.

See Fig. 22 for the implementation of one of the (simpler) explication transformations defined in the Stratego implementation. This transformation, called *auto-value*, gives every state a value if it does not (explicitly) have one. The rule defined on line 1 traverses top-down over the AST to try and apply the rule `oil-auto-value-term` on every node. This rule, defined on lines 3–5, does the actual explication: if a state without a value is found (line 4), the value is added (line 5). It uses the rule defined on line 7, which creates a fresh name for the state value given the name of the state.

6.3 Resilient staging

To keep the desugaring transformations simple, they each have expectations on the input. For instance, desugaring transformation *auto-type* that derives new types from the areas of an OIL specification expects that each state has a value. Most of these expectations are ensured by other desugaring transformations. For instance, *auto-value* ensures that every state has a value, which matches the expectation of *auto-type*.

To help us structure the desugaring transformations, as well as the transformation architecture as a whole, a framework that we call *resilient staging* is used. This framework is based on *stages*, which are equipped with a precondition, a transformation, and a postcondition. Each stage should only have one specific transformation purpose to keep them well maintainable and reusable. Stages can be concatenated to create larger transformations, which we call *pipelines*.

The precondition represents requirements on the input of the stage, such as the presence or absence of specific terms or term patterns. When executing a stage, the precondition is checked first. If the precondition is met, the actual transformation will be executed. Otherwise, the pipelines stops and reports the errors from the precondition. For some stages, checking the precondition may require work that is useful for the transformation itself too, such as collecting specific terms. To prevent duplicate work, the precondition may also pass data to the transformation if the precondition is met.

After the transformation has been executed, the postcondition is checked. This postcondition represents requirements on the output of the stage, effectively testing whether the transformation was successful. If the postcondition is not met, the pipeline is aborted and errors are returned. Ideally, postconditions checking should only be enabled during development, since stages should be correct when used in production.

In a sense, the pre- and postcondition provide a contract over the transformation: they define what is required by the transformation and what can be expected from the transformation. A clear contract and transformation purpose indicate how and where transformations should be embedded into pipelines, also promoting reusability. When a transformation is used or defined incorrectly, the stage conditions will show what and where the issue is, hence "resilient" in resilient staging.

In Stratego, stages are defined using the transformation rule `stage` shown in Fig. 23. The three parameters `pre`, `trans` and `post` are the precondition, transformation and postcondition, respectively. The precondition and postcondition are transformations too, which return a list of errors, given an input AST. More on this in Sect. 7.1. Whenever a condition returns errors, the pipeline is abandoned and the errors are returned. See Fig. 24 for the instantiation of the stage of *auto-value*.

```
stage(pre, trans, post):
    StageSuccess(ast) -> result
  where
    //check preconditions
    (pc-data, errors) := <pre> ast;
    if (<?[]> errors) then

      //do the transformation
      ast' := <trans> (pc-data, ast);

      //check postconditions
      errors' := <post> ast';
      if (<?[]> errors') then
        result := StageSuccess(ast')
      else
        result := StageFailure(ast', errors')
      end

    else
      result := StageFailure(ast, errors)
    end
```

Fig. 23 The Stratego transformation rule to define a stage (simplified)

```
oil-auto-value-stage =
  stage(
    stage-preconditions-true,
    oil-auto-value,
    all-states-value
  )
```

Fig. 24 The creation of the stage for *auto-value* in Stratego, using a generic rule `stage-preconditions-true`, the transformation from Fig. 22 and a postcondition rule `all-states-value`

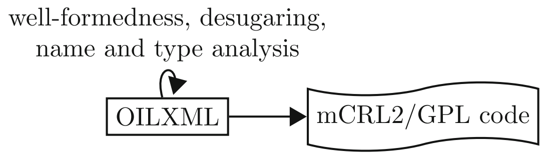

Fig. 25 An overview of the transformation architecture of the implementation of OIL in Python. The rectangular box is an AST schema, the wavy box is text, and the arrows are transformations

6.4 The Python implementation

In the Python implementation, no intermediate representations are used. The representation that results from parsing OILXML is used directly for checks and transformations. See Fig. 25 for the general transformation architecture. This representation consists of two parts: the AST representation generated by the Minidom parser and the AST representation generated by the expression parser. For the Minidom AST representation, many helper functions have been defined that hide the use of Minidom, mainly for the access or derivation of information from it. Many of such information is cached so that the Minidom AST does not need to be accessed too frequently. For the expression AST representation, a custom `Expression` class is defined that is used to represent any expression term.

```
 1  def addAutoStateValues(spec):
 2      dom = spec.getDom()
 3      for state in spec.getElements('state'):
 4          if len(Children(state, ['value'])) == 0:
 5              stateName = state.getAttribute('name')
 6              autoValue = dom.createElement('value')
 7              literalName =
    spec.createUniqueLiteralName(stateName)
 8              _setExpressionText(spec, autoValue,
    f"'{literalName}'")
 9              state.insertBefore(autoValue,
    state.firstChild)
10              state.hasAutoTerm = True
11      spec.initElementCache()
```

Fig. 26 The Python implementation of the *auto-value* desugaring transformation

Desugaring transformations are defined as Python functions that traverse the AST and apply the changes where necessary. See Fig. 26 for the implementation of *auto-value* in Python. It traverses the AST to find all states (line 3). Then if this state does not have a value (line 4), a new value name is created based on the name of the state (lines 5 and 7), a new AST element is created that holds this value (lines 6 and 8), which is then added to the state (line 9).

The Python implementation does not explicitly define pre- and postconditions per transformation like resilient staging does, but it does define transformations and conditions on the AST as separate functions, which are called in a specific (interleaving) order.

6.5 Evaluation

To evaluate the productivity of implementing abstract syntax, we look at two evaluation points: AST representations and desugaring transformations.

6.5.1 AST representations

Question. Does it cost less code volume to define AST representations for OIL in Spoofax compared to Python?

Method. We collect all AST representations that are used in the transformation architecture of an OIL specification in both the Spoofax and the Python implementation and measure the SLOC used to define them.

Results. As shown in Fig. 20, the Spoofax implementation defines seven AST schemas for OIL: OILXML AST, OILDSL AST, the three IRs, mCRL2 AST and GPL AST. As shown in Fig. 25, the Python implementation defines one AST representation. This AST representation is split into two parts: a Minidom AST representation and an expression AST representation.

In the Spoofax implementation, all AST schemas, apart from the normalized and desugared IR, are automatically derived from their grammar defined in SDF3. The normalized and desugared IR do not have their own grammar and have their constructors defined in Stratego instead. Their AST schemas are defined with 43 SLOC and 23 SLOC, respectively, with a shared expression AST schema defined with 33 Stratego SLOC. In the Python implementation, the constructors for the Minidom AST representation are automatically derived from the Minidom parser. The `Expression` class used for the expression AST representation is defined with 9 Python SLOC.

Analysis. If a grammar already exists, the Spoofax implementation does not need any SLOC to define the AST schemas as they are generated automatically. This is also the case in the Python implementation for the Minidom AST representation. If the grammar is not available, an AST schema can be defined in Spoofax with 1 Stratego SLOC per constructor, as is the case for the normalized and desugared IRs. In Python, the expression AST representation only consists of one constructor, namely a general `Expression` constructor with 7 children, defined with 9 SLOC. Due to this constructor, the expression AST representation in Python is more general than the expression AST schema in Spoofax, as the latter explicitly defines a constructor for each type of expression. This is also the reason why the expression AST schema in Spoofax uses more SLOC than the expression AST representation in the Python implementation.

Conclusion. Due to the differences in available AST representations, we cannot give a conclusion on the definition of AST representations. For the AST representation that is available in both implementations, namely that for expressions, we cannot draw a conclusion either, due to the different approach for constructors and the small size.

Discussion. The difference between the Spoofax and Python implementations in the use of IRs is partially due to the mutability of ASTs. ATerm terms are immutable, so any desired change to the definition of an AST requires the definition of new constructors. In the Python implementation, the AST is mutable, so the AST can be changed dynamically. This does have the downside that it is more difficult to know exactly what information is available at any point in the transformation architecture. Due to the mutability of the AST in the Python implementation, it was found during its development that the definition of explicit IRs would not be worth the effort for the benefits it could bring over using the single mutable AST.

An interesting observation is the difference in the approach of defining the expression AST schema between Spoofax and Python. In Spoofax each type of expression is defined with an explicit constructor, while the Python implementation uses one generic constructor. This is related to the transformation language available that operates on the ASTs. In Spoofax, Stratego is used for transformations, where the terms and their constructors are part of the data language. Having concisely represented terms therefore also helps keeping transformations concise. In Python, it is more common to

Table 2 SLOC for the implementation of desugaring transformations that are defined in both the Spoofax and Python implementation of OIL

artifact	Stratego	Python
Distribute-groups	41	41
Auto-region	16	31
Auto-super	31	37
Auto-state	127	94
Auto-value	5	34
Auto-variable	39	52
Auto-type	26	31
Auto-init	15	38
Auto-silent	9	18
Reused	42	251
Total (Without reused)	309	376
Total (All)	351	627

reason in terms of classes with attributes. Since all expression types have similar attributes, such as a list of subexpressions, it makes more sense to define one generic constructor.

Though the Minidom AST representation in the Python implementation is automatically generated, thanks to the Minidom library, this library can only be used for XML-based languages. It can be expected that for non-XML languages much more effort is needed to define an AST representation, which is also the reason why the implementation does not define IRs. In Spoofax, AST schemas can be defined for any textual language in an equally productive way.

6.5.2 Desugaring transformations

Question. Does it cost less code volume to define the desugaring transformations for OIL in Stratego compared to Python?

Method. We collect all desugaring transformations that are implemented in both the Spoofax and Python implementation and measure the SLOC used to implement them. Any SLOC called by the desugaring transformations, such as helper functions, are counted too.

Results. Table 2 shows the SLOC used to implement the desugaring transformations in both implementations. Any SLOC that are used for more purposes than one desugaring transformation are captured in the "Reused" row. The body of reused code in the Python implementation mainly consists of helper functions for traversal through the Minidom AST representation or for general simple transformations. The Stratego implementation has less reused code, because most of such traversals or simple transformations can be compactly expressed with the Stratego language and its standard library.

Not every desugaring transformation in this table corresponds to one transformation in Stratego and one function in Python. Desugaring transformation *distribute groups* corresponds to two Stratego transformations and one Python function, *auto-region*, *auto-variable*, *auto-type* and *auto-init* correspond to one Stratego transformation and two Python functions, and *auto-state* corresponds to two Stratego transformations and three Python functions.

Analysis. In total, the desugaring transformations are implemented with Stratego with a factor of 0.56 SLOC compared to Python. On average, not counting reused SLOC, a desugaring transformation is implemented with Stratego with a factor of 0.82 SLOC compared to Python. This difference is mainly due to the fact that Stratego is specifically tailored for transformations, it allows one to define transformations in a more compact way. Stratego does this with ATerm as the main data format and with the core support for pattern matching.

We take the implementation of *auto-value* as an example. Due to ATerm as data format, one can create the resulting state node by writing the resulting term as the right-hand side of a transformation rule (Fig. 22 line 5), instead of creating the new node and text fields step by step (Fig. 26 lines 6–9). Due to the core support for pattern matching, checking whether the state has no value and assigning the name of the state to a variable can be done by just writing the `State` term with required subterms (such as `None()`) and variables (such as `name`) as the left-hand side of a transformation (Fig. 22 line 4), whereas in the Python implementation this is done in separate steps (Fig. 26 lines 4–5). For this example, the default support in Stratego for creating a fresh name (`newname` in Fig. 22 line 7) also helps significantly, as this is implemented explicitly in the Python implementation with a (not reused) function of 23 SLOC (called in Fig. 26 line 7).

Deviations from the average SLOC ratio that are more in favor of the Python implementation are typically in case of transformations that require less local changes, such as *auto-region*, *auto-super*, *auto-state* and *auto-variable*, which introduce regions, zones, states and variables, respectively, based on multiple sources of information in the AST. This is for a few reasons. First of all, it is not possible in Stratego to access the parent of a term directly. To access such information, a top-down traversal is needed that keeps track of previously encountered terms. Secondly, it is not straightforward in Stratego to store information globally to reuse later, which is for instance necessary to make sure that all names created are distinct. It is possible to use dynamic rules [58] for this, but the dynamic transformation behavior that these rules add makes it more difficult to understand how Stratego code executes when reading it. Lastly, the immutability of ATerm makes it impossible to store references to parts of the AST. First collecting information and then transforming this information does not have an effect on the resulting AST; instead, the information needs to be mapped back to the AST to then transform it, or information collection and transformation should be intertwined. In Python, these three restrictions of Stratego are less of a concern due to the freedom one has in a general-purpose programming language.

An example type of operation used in desugaring that is affected by the restrictions of Stratego above is finding the least common ancestor (LCA) of two areas. In the Python implementation, this is done by walking up the AST from the two areas using parent pointers until both paths cross. In Stratego, such a walk along parent pointers is not possible. Instead, the complete AST is traversed bottom-up in a recursive fashion, where for each area in the tree all descendant areas are collected. If the two areas for which the LCA needs to be found are in this collection for the first time, the LCA is found. An implementation for finding the LCA has similar SLOC between Stratego and Python, but the implementation in Python is reused between multiple transformations, while the implementation in Stratego is not as it is part of the basis of the desugaring transformation.

The only difference in functionality between the two implementations are some naming conventions for newly introduced elements. For instance, in *auto-value*, the Stratego implementation creates a name that is different from any name in the OIL specification, whereas the Python implementation creates a name that is different only from all literals in the OIL specification. This choice is sufficient, but requires one to specifically collect all literals, which contributes to 21 SLOC of *auto-value* in the Python implementation.

Conclusion. Although there are some restrictions to Stratego that hinder the conciseness of transformations defined in it compared to Python, on average Stratego requires less code volume compared to Python to define the desugaring transformations.

Discussion. As mentioned in the analysis, there are some restrictions to Stratego due to limitations on what it supports when compared to Python, such as not being able to directly access the parent of a term. We do not necessarily see these restrictions as points of improvement however. For instance, while the immutability of the AST may make it impossible to store references to terms in the AST, which restricts the design of transformations, this does make sure that an AST cannot be transformed indirectly, which is beneficial for the understanding of Stratego code.

The measurements only consider the SLOC used to implement the functionality of the desugaring transformations, not the composition of them. In the Spoofax implementation, they are composed using the resilient-staging framework. This framework is defined with 63 SLOC, which can be reused for any language and any transformation architecture. Creating a stage from a desugaring transformation then costs one `stage` transformation rule call, as shown in Fig. 23. The stages are then sequentially composed. Since the Python implementation does not implement resilient staging explicitly, it does not have this overhead.

7 Static semantics

In this section, we discuss the implementation of static semantics of OIL in Spoofax, which we consider to consist of name binding, typing and other well-formedness aspects, which together realize OF3 (Input Correctness). Name binding and typing are implemented using NaBL2. Well-formedness is mostly implemented with a collection of Stratego transformations that produce errors if the constraints are violated. In the transformation architecture of OIL, well-formedness checking occurs on the normalized IR, while name binding and typing occur on the desugared IR (see Fig. 20).

We describe how transformations and origin tracking are used to realize well-formedness checking and how cross-file and cross-language analysis over a collection of IDL and OIL files is realized, which relates to OF4 (Language Interaction). Afterward, we discuss how static semantics is realized in the Python implementation. We then evaluate Spoofax on productivity in the context of static semantics.

7.1 Well-formedness checking

OIL specifications need to conform to well-formedness constraints. For example, each region should have at least one state. Although this particular example could have been enforced in the grammar, not all well-formedness constraints can be enforced in a grammar, or they lead to messy grammars. Also, by implementing these constraints manually, it is possible to generate better error messages than generic errors generated by the parser.

For some well-formedness constraints, such as illegal variable names and name distinctness, there is core support in SDF3 and NaBL2 respectively. Other constraints are checked by means of Stratego transformations, which transform an AST to a list of errors. Figure 27 depicts a Stratego rule that implements a well-formedness constraint that says that every state must have a value. This check is used as the postcondition of the stage of desugaring transformation *auto-value*, see Fig. 24. If the check fails, the rule returns a list of errors,

```
all-states-value =
  collect-all(\
    s @ State(_, None(), _, _) ->
    (s, "State does not have a value")
  \)
```

Fig. 27 A Stratego code snippet that checks whether all states have a value

```
1  init ^ (s) :=
2    ...
3    new s,
4    distinct/name D(s)/Module | error
       "Duplicate module" @NAMES.
5
6  [[ IDLModule(m, imps, defs) ^ (s) ]] :=
7    Module{m} <- s,
8    ...
```

Fig. 28 A NaBL2 code snippet that checks duplication of module names

one for each state for which the check fails. Each error is a tuple that contains both the state term and the error message.

Although the constraint is defined on an IR—and thus not on the original parsed AST—Spoofax can associate the error with the original input syntax. This is thanks to origin tracking. The origin information of a term, that is created when a specification is parsed, is passed on when this term is transformed into another term. This makes the origin information directly accessible from the term within an error tuple. Spoofax can then use these error tuples to show the error to the user on the correct syntactical element in the editor.

7.2 Cross-file and cross-language analysis

The implementation of OIL and IDL in Spoofax involves static analysis that spans both multiple files and multiple languages. For instance, for a collection of multiple IDL files it should be checked whether there are no modules defined with the same name (cross-file analysis, within the same language). As mentioned in Sect. 3.2, transitions in OIL refer to operations defined in IDL files (cross-language analysis). We discuss how both forms of analysis are implemented using NaBL2.

Figure 28 depicts an NaBL2 code snippet that enforces the cross-file constraint that no modules with duplicate names may exist across IDL files. Line 1 defines the `init` rule, which is where the analysis starts. NaBL2 is configured such that the scope `s` that is created in the `init` rule (line 3) is used as the initial scope for each IDL file. For IDL files, this scope is supplied to the rule for `IDLModule` (line 6), which adds a declaration of the module to this scope. By attaching the scopes of all IDL files to the single root node, all IDL modules are part of a single scope graph. The restriction that all module names are distinct is defined on line 4, where `D(s)/Module` defines the collection of all `Module` elements reachable from scope `s` and `distinct/name` defines that no two elements in this set may have the same name.

Figure 29 depicts an NaBL2 code snippet that specifies the cross-language importing of IDL modules into OIL specifications. First a reference to the module is added to the scope

```
1  [[ DESImportModule(m) ^ (s) ]] :=
2    Module{m} -> s,
3    Module{m} |-> decl | error "Module not
       found" @ m,
4    Module{m} <=== s.
```

Fig. 29 A NaBL2 code snippet that imports IDL modules into OIL

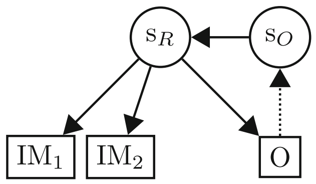

Fig. 30 An abstract representation of a merged scope graph of an OIL specification (O) and two IDL modules (IM_i). Rectangles indicate declarations, circles indicate scopes. S_R is the root scope. The scope associated with the OIL specification (S_O), indicated with the dotted arrow, has the root scope as its parent scope, thereby making the IDL modules visible from the OIL specification

(line 2), after which it is checked whether the referenced module can be found (line 3), that is, whether a path from the reference to the declaration exists in the scope graph. Then in line 4 all declarations in the module are imported. More precisely, `Module{m}<=== s` makes all declarations that are visible in the scope on which `Module{m}` was declared visible in `s`.

Similar to how multiple IDL files share a single scope graph, the scope graphs of OIL files could conceptually be connected with those of IDL files to import the analysis for importing of Fig. 29. Figure 30 depicts this. However, in Spoofax it is not possible to implement this directly. Spoofax only supports configuring NaBL2 to have analysis span multiple files of a single language, but not the files of multiple languages. This has been worked around by instantiating a single language artifact that accepts both IDL and OIL files. Although the implementation sources of IDL and OIL are organized in separate projects, there is no distinction anymore between an IDL and OIL language artifact; for IDL modules to be usable in OIL specifications, they have to be in files with the same `.oil` extension.

7.3 The Python implementation

In the Python implementation, typing of expressions is implemented with a bottom-up recursive algorithm with a case distinction on the type of operator. For some operators, such as equality, the process is repeated but with an expected type. Name resolution is partly done by the type checker and partly by separate functions. Well-formedness constraints are defined with separate functions.

See Fig. 31 for how name resolution of import statements is done in the Python implementation. On line 2, information

```
 1  def extractModules(spec):
 2      idlInfo = spec.generator.getIDLInfo()
 3
 4      rootModules = OrderedSet()
 5      for _import in spec.getElements('import'):
 6          moduleName = _import.getAttribute('module')
 7          if moduleName in idlInfo:
 8              rootModules.add(moduleName)
 9          else:
10              spec.addAttributeValueError(_import,
    'module', None, 'Module not found in "idl_specs.txt"
    nor in the additional IDL include directory.')
11      spec.rootModules = rootModules
```

Fig. 31 A Python code snippet that imports IDL modules into OIL

on the IDL files is retrieved. This loads the relevant IDL files and creates a dictionary representing them, if this was not done already. Then on lines 5–7 the function iterates over all module names that appear in import statements. It checks on line 7 if this module name exists and if not, it reports an error (line 10). On line 11, the list of imported modules is stored in the OIL specification object for easy access.

Whereas the Spoofax implementation aborts the pipeline for any stage pre- and postcondition that fails, the Python implementation can do multiple steps before aborting. When to abort in case of errors is decided manually, by means of conditional return statements throughout the sequence of desugaring transformations and well-formedness checks. Errors can be shown in the web interface of the Python implementation on a textual representation of the OIL specification thanks to origin tracking. Syntactical elements with errors are highlighted in red and hovering over them shows an error message.

7.4 Evaluation

To evaluate the productivity of implementing static semantics, we look at a single evaluation point: the implementation of static semantics in OIL's implementations in Spoofax and in Python. We consider five static semantics artifacts: name binding, typing, well-formedness, error handling and error reporting.

Question. Does it cost less code volume to define the static semantics artifacts for OIL in Spoofax compared to Python?

Method. For the name binding, typing and well-formedness artifacts that are implemented in both Spoofax and Python, we measure how many SLOC were used to implement them. We also measure the SLOC used to abort when errors are found (error handling) and the SLOC used to show the errors to the user (error reporting). Any code called by name binding, typing and well-formedness definitions, such as helper functions, are counted too.

Results. Table 3 shows the SLOC used to implement name binding, typing and other well-formedness, as well as error handling and error reporting in both Spoofax and Python. Only name binding and typing over syntactic elements that are defined in both the Spoofax and the Python implementation are considered. Any SLOC that are relevant for more artifacts than just one out of name binding, typing and well-formedness are captured in the "Reused" row. All SLOC under Spoofax name binding, typing and reused are written in NaBL2. More specifically, NaBL2 code that only relates to creating and querying the scope graph corresponds to name binding and NaBL2 code that only relates to type definitions and type checking corresponds to typing; the rest corresponds to "Reused". Well-formedness in Spoofax is a combination of SDF3, NaBL2 and Stratego. Error handling and error reporting are defined in Stratego.

Table 3 SLOC for the implementation of name binding, typing and other well-formedness over the syntactic constructs that are defined in both the OIL and Python implementation

Artifact	Spoofax	Python
Name binding	128	233
Typing	81	257
Well-formedness	24	57
Error handling	15	29
Error reporting	19	37
Reused	208	298
Total (Without reused)	267	613
Total (All)	475	911

To create a fair comparison, we have not counted any SLOC that produces functionality that is not in the other implementation. For the name binding and typing in Spoofax written in NaBL2, this meant that 88 SLOC was not counted. This 88 SLOC includes analysis of syntactic elements not implemented in Python and typing of elements that is not done in Python, such as areas and operations. For the name binding and typing by the type checker of the Python implementation 101 SLOC was not counted, which consists of analysis of syntactic elements and other checks not done in the Spoofax implementation. Since the well-formedness constraints are implemented as separate rules in Spoofax or functions in Python, we can measure them separately. Only a few well-formedness constraints have been measured, since many do not correspond well to any constraint in the other implementation.

Analysis. As Table 3 shows, name binding, typing and other well-formedness are implemented in Python in about double the SLOC compared to Spoofax. In general, the lower SLOC for Spoofax can be explained by the fact that the meta-DSLs that are used, especially NaBL2, are specifically made for the implementation of these aspects.

Only looking at SLOC specific to name binding, the Spoofax implementation uses a factor of 0.55 SLOC compared to the Python implementation. In NaBL2, SLOC specific to name binding consist of creating and querying the scope graph. In Python, this involves reading in IDL files, creating classes for easy access to information in IDL files, checking name binding by querying this information and that of the OIL specification, and adding name binding information to the AST. The last two cause the main difference in SLOC between NaBL2 and Python. Checking name binding in NaBL2 is done by adding the reference to the scope graph and then checking for a path to the declaration as shown in lines 2–3 in Fig. 29. How this declaration is found does not need to be implemented explicitly, while in Python all declarations are retrieved manually to explicitly check whether the relevant declaration exists. Adding name binding information to the AST also needs to be explicitly implemented in Python, while this implicitly happens in NaBL2 by having a constraint rule for every term.

Looking only at typing-specific SLOC, the Spoofax implementation uses a factor of 0.32 SLOC compared to the Python implementation. For both implementations, most of the SLOC are in the typing of expressions. One of the main reasons that the Python SLOC is higher than the NaBL2 SLOC is that in Python there are some binary operators, such as equality and assignment, for which many case distinctions are defined based on the types of operands they can have. In NaBL2 these case distinctions are not necessary as they happen implicitly.

Looking only at well-formedness-specific SLOC, the Spoofax implementation uses a factor of 0.42 SLOC compared to the Python implementation. This is partly due to core support for some specific forms of well-formedness in Spoofax, such as rejecting specific variable names and checking for distinctness of names within a scope. Such checks only take 1 SLOC in SDF3 and NaBL2, respectively, see Fig. 28 for an example, whereas in Python these require explicit traversal of the AST. Other well-formedness constraints in Spoofax are implemented in Stratego, for which the same productivity conclusions hold as for desugaring transformations (Sect. 6.5).

For error reporting and error handling, the Spoofax implementation uses about half the SLOC compared to the Python implementation. The difference in error handling SLOC is because the Spoofax implementation has a generic way of aborting pipelines built into the resilient-staging framework, while in the Python implementation abort points are places manually, which produces code duplication.

Both implementations support two types of error reporting: by means of highlights on the original specification or by means of a list of errors. For the first, the Spoofax implementation only needs minor general configuration, while the Python implementation explicitly locates and colors the syntactical elements in an HTML generator specific for XML-based languages. For the second, both implementations support a generic way of displaying the list of errors.

For both implementations, a large portion of the SLOC are reused. The reused SLOC in Spoofax is only relevant for static analysis: 174 SLOC consists of constraint rule declarations that define how name binding and typing information is related between a term and its subterms, and 34 SLOC is related to naming and importing modules. The reused SLOC in Python consists mainly of parts of the type checker that also involve variable reference resolution, as well as helper functions that retrieve information from the OIL and IDL specifications, which are also used for more purposes than only static semantics.

Conclusion. For static semantics, the Python implementation uses about twice the amount of SLOC compared to the Spoofax implementation, for the same functionality, with or without considering reused SLOC. This is mostly due to the fact that NaBL2 is specialized for name binding and typing and because of core support for specific types of well-formedness. Error handling and reporting can also be defined in a more concise and generic way. This shows that it costs less code volume to implement the static semantics artifacts for OIL in Spoofax compared to Python.

Discussion. Like with the concrete syntax definition, the name binding and type checking of IDL and OIL in the Spoofax implementation are split up into multiple NaBL2 modules, 21 in total. Unlike with the concrete syntax definition, NaBL2 files do not need import statements to compose them. All NaBL2 files in a project are deemed relevant and are collected automatically, so no SLOC is needed to compose NaBL2 modules. An exception to this is the composition of OIL-specific analysis with IDL analysis, which costs about 12 SLOC. This includes project configurations to export the IDL NaBL2 definitions as well as imports of IDL type signatures and files generated from the IDL NaBL2 definitions.

The workaround to make it possible to do name resolution between IDL and OIL files did not cost any extra NaBL2 SLOC. It did, however, cost 26 extra Stratego SLOC (not in Table 3) to define transformation rules that check whether the given AST is an IDL or OIL specification, which are needed at the beginning of end-to-end transformations.

A big reason for the conciseness of NaBL2 is that it is a declarative language, which means that one does not implement how the program executes. This can, however, also make it unpredictable how the NaBL2 analysis is executed. For example, when a reference of an integer type is used at a location where a Boolean is expected, the type error could be reported on the declaration of the integer variable, while the error is expected on the reference. Although NaBL2 specifications can be annotated to indicate a preference for reporting the error on the declaration, this does not cover all cases.

The implementation of name binding and typing in Spoofax also automatically provides some editor services. When hovering over a syntactical element in the editor, its type is shown in a small text box. Also, navigating through a reference moves the cursor to the corresponding declaration, even if both are in different files. The Python implementation does not provide these editor services.

8 Dynamic semantics

OIL is a language for defining the behavior of control software. What the actual behavior is of an OIL specification is described by its dynamic semantics, which is formally defined in [50]. This semantics is implemented in Spoofax using Stratego with two code generators: one for verification (mCRL2) and one for execution (C++). These together realize OF5 (Multiple Targets).

Like OIL, mCRL2 is a language for describing system behavior, except that it is based on process algebra [45]. It also comes with a toolset [59], containing all kinds of model checking functionalities, such as checking properties and checking behavioral equivalence, as well as tools to simulate and visualize the behavior of an mCRL2 specification. With this translation from OIL to mCRL2, the functionality of the mCRL2 toolset can be indirectly used for OIL specifications as well. Some early results of this were already presented in [50].

To actually use an OIL specification to implement a software system, executable code needs to be generated. For that reason, a translation from OIL to C++ was implemented. This translation is inspired by the C++ generator in the Python implementation, which was already used for some systems in development at Canon Production Printing.

We highlight three parts of the implementation of these two translations in Spoofax: how the implementation of dynamic semantics is split into many projects, how static analysis results are queried for use in transformations and how configurability of a translation is handled. Afterward, we discuss the implementation of these translations in the Python implementation. We then evaluate Spoofax on productivity in the context of dynamic semantics.

8.1 Division into projects

As is already discussed in Sect. 6 and shown in Fig. 20, an OIL specification is first transformed to the semantic IR in the Spoofax implementation before it is transformed into mCRL2 or GPL code. For the sake of extensibility, the semantic IR, mCRL2 and GPL are all defined in separate projects, as well as the transformations between them. See Fig. 32 for the hierarchy of these projects, where the SEM project defines the semantic IR. All projects in this hierarchy are part of the Spoofax implementation except for the mCRL2 project, which already existed for other purposes.

The translation to GPL defined in OIL2GPL does not translate directly to C++, but to an intermediate representation called the GPL IR first instead. The GPL IR is a pseudo code representation defined in the GPL project with basic object oriented imperative programming language constructs such as classes, methods, basic statements and expressions. The GPL project then defines a translation from the GPL IR to C++ files. The reasoning behind the creation of the GPL IR is to make it relatively simple to add translations to other general-purpose programming languages: only a transformation from the GPL IR to that programming language needs to be implemented.

See Figs. 33 and 34 for how the GPL IR splits up the transformation of an enum declaration to C++. In Fig. 33, an enum type declaration of the semantic IR is transformed to an enum type declaration of the GPL IR, which changes the type name using the name of the OIL specification and the original type name. In Fig. 34, for every enum type declaration (line 3), a C++ enum class is created (line 8–10). While most transformations in the Spoofax implementation of OIL are model-to-model transformations, the transformation from the GPL IR to C++ is a model-to-text transformation, as can be seen by the use of templates. This way, it is not necessary to define the C++ syntax in Spoofax.

8.2 Using static analysis results

When NaBL2 name binding and typing have been applied on an AST, all terms in the AST are annotated with information that stores the results of the analysis. This information can then be used in Stratego transformations by means of specific transformation rules. For instance, the type of a term can be extracted with the rule `nabl2-get-ast-type`. See

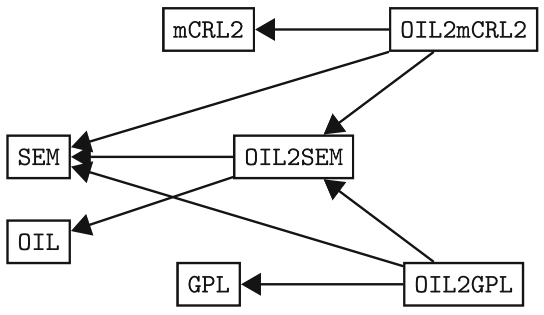

Fig. 32 A project extension graph of projects used for code generation in the Spoofax implementation. Boxes correspond to projects. Arrows mean “extends”

```
oil-sem2gpl-spec-enumdef(|specName):
  SEMEnumDef(type, items) ->
  GPLEnumDef(<SCOPEDTYPE(|specName, type)>,
    items)
```

Fig. 33 The Stratego transformation of enum declarations from the semantic IR to the GPL IR

```
gpl2h: GPLProgram(_, ..., enumDefs, ...) ->
  $[...
  [<join-strings(|"\n\n")>
    <map(gpl2h-enumDef)> enumDefs]
  ...
  ]

gpl2h-enumDef: GPLEnumDef(name, items) ->
  $[enum class [name] {
  [<indent-text(|2)> <join-strings(|",\n")>
    items]
  };]
```

Fig. 34 The Stratego transformation of enum declarations from the GPL IR to C++

```
oil2mcrl2-type =
  nabl2-get-ast-type ; oil2mcrl2-map-type
oil2mcrl2-map-type : TInteger -> Int()
oil2mcrl2-map-type : TBool    -> Bool()
```

Fig. 35 A Stratego transformation rule that transforms a term to one that represents its type in mCRL2

```
1  [[ e@EnumDef(name, literals, ...) ^ (s) ]] :=
2      Type{name} <- s,
3      Type{name}.decl := e,
4      enum_ty == TEnum(Type{name}),
5      Type{name} : enum_ty !,
6      ... ]]
7
8  TypeRef [[ TypeReference(x) ^ (s) : ty ]] :=
9      Type{x} -> s,
10     Type{x} |-> d | error $[Type [x] not
       found] @ x,
11     d : ty.
```

Fig. 36 Two NaBL2 code snippets: one that stores the declaration of an enum type inside its type (simplified) and one that defines type inference for type references

Fig. 35 for a (partial) definition of a transformation rule that uses this rule. Given a term with type information, such as a variable reference term, it returns a term that represents its type in the mCRL2 AST schema. The terms starting with `T` correspond to the type as annotated by NaBL2.

It is possible in NaBL2 to annotate the AST with more information than the default name binding and typing results. Scope graph nodes can be given *properties*, which can store any term. We use this, for instance, for retrieving the declaration of an enum type, defined in an IDL specification, when we generate code for an OIL specification that has a reference to this enum type. See Fig. 36 for an example, where the declaration of an enum type in an IDL specification is stored in its scope graph declaration node `Type{name}` with a property `decl` (line 3). This declaration node is then

```
1  sem2mcrl2-imported-enum-typedef:
2    ref -> ...
3  where
4    a := <nabl2-get-ast-analysis> ref;
5    TEnum(occ) := <nabl2-get-ast-type> ref;
6    EnumDef(type, items, _) :=
       <nabl2-get-property(|a, "decl")> occ;
```

Fig. 37 A Stratego transformation rule that uses the scope graph node stored in the type of an enum variable reference to obtain the declaration of the corresponding enum type

stored within the `TEnum` type `enum_ty` of the enum (line 4). Whenever a reference to this enum type in the OIL specification is encountered (line 8), we add a reference node to the scope graph (line 9) and try to resolve it (line 10) like discussed in Sect. 7.2. If successful, the `TEnum` type `ty` of the enum type reference (line 8) is inferred from the enum type declaration `d` by requiring that `d` also has type `ty` (line 11). Since property `decl` is stored within this `TEnum` type, the property becomes available in the context of the enum type reference in the OIL specification.

See Fig. 37 for a Stratego transformation in which the property is retrieved from an enum type reference, which is part of the translation from an OIL specification to mCRL2. First the type of the type reference is extracted (line 5), after which the `decl` property is queried on the node within the type, which provides the enum type declaration (line 6). This enum type declaration can then be translated to one in mCRL2 (line 2, details not shown).

8.3 Configurability of the mCRL2 generator

The translation to mCRL2 has a number of configuration options. Some of these options are mainly useful for debugging the translation during development, but other options result in a significantly different output. For instance, one option mainly used for debugging is whether to use auxiliary variables in the generated mCRL2 specification that help enhance its readability. Since these options change how the mCRL2 specification should look like, they configure the transformation that generates mCRL2. One way to implement this in Stratego is by passing the configuration information on as parameters of transformation rules. However, the more complex a transformation becomes and the more deeply nested this information is used, the more cluttered with such configuration parameters the transformation becomes. A solution in many languages would be to define global variables that hold this information, but Stratego does not support global variables.

Instead, *dynamic rules* [58] are used. A dynamic rule is a transformation rule that is created during transformation time, whose behavior can depend on the status of the trans-

```
oil2mcrl2-dyn-options(aux-vars) =
  (aux-vars < rules(mcrl2-aux-vars : t -> t)
   + rules(mcrl2-aux-vars : _ -> <false>))
```

Fig. 38 The creation of dynamic rules in Stratego for the auxiliary variables configuration

```
sem2mcrl2-process-trans-pre =
  if mcrl2-aux-vars then sem2mcrl2-firedvar
    else sem2mcrl2-trans-pre end
```

Fig. 39 An example Stratego transformation rule where the dynamic rule `mcrl2-aux-vars` is used

formation at that time. Such rules are used for configuration by creating a dynamic rule for each value of a configuration option, which always succeeds if the value was chosen, otherwise it always fails. For Boolean configuration options one dynamic rule suffices. These rules can then be used wherever the differences between configurations have effect on the transformation, without having to pass anything on explicitly.

See Fig. 38 for a Stratego rule that creates the dynamic rule `mcrl2-aux-vars` for the auxiliary variables configuration, which is done just before the transformation from the semantic IR to mCRL2 is applied. The parameter `aux-vars` stores whether the user has chosen to introduce auxiliary variables. This parameter is then used in a ternary operator of the shape `s1 < s2 + s3`, which acts similarly to an if-then-else. If `aux-vars` is true, the dynamic rule `mcrl2-aux-vars` is defined as a rule that always succeeds (`t -> t`), else as one that always fails (`_ -> <false>`). See Fig. 39 for an example where this dynamic rule is used during the transformation to mCRL2. If the user chose for the introduction of auxiliary variables, the precondition of a transition in the mCRL2 process should be represented with an auxiliary variable (`sem2mcrl2-firedvar`), otherwise the full transition precondition is used (`sem2mcrl2-trans-pre`).

8.4 The Python implementation

The Python implementation also defines a translation to mCRL2 and a translation to C++. The translation to mCRL2 in the Python implementation was created during an exploratory study on the semantics of OIL and is therefore only a prototype. Compared to the Spoofax mCRL2 generator, the Python mCRL2 generator supports slightly fewer OIL language constructs and it can only generate mCRL2 code for single components, whereas the Spoofax mCRL2 generator can also generate mCRL2 for systems of components. On the other hand, the Python C++ generator has been maintained and refined for years and has been used to generate C++ for systems used in production. Compared to the Spoofax

```
1  localEnumTypes = spec.getLocalEnumTypes()
2  for tName in localEnumTypes:
3      tgt.nl()
4      tgt <<= f'enum class {Camel(tName)}'
5      tgt <<= '{'
6      tgt.indent()
7      tgt <<=
       (f',{tgt.eol}{tgt.ind}').join(localEnumTypes[tName])
8      tgt.dedent()
9      tgt <<= '};'
```

Fig. 40 The Python transformation for enum declarations

C++ generator, the Python C++ generator supports slightly more OIL language constructs and it is built to fit into Canon Production Printing's software base. This includes adherence to coding standards and a higher level of configurability of the generated C++ code, such as allowing multiple types of schedulers to execute the specification, which is not supported in the Spoofax C++ generator.

Both the Python mCRL2 generator and the Python C++ generator are defined in their own files in the Python implementation. Since the Python implementation does not have any explicit IRs, both generators directly transform the (desugared) OIL specification to the desired target. See Fig. 40 for an excerpt of the Python C++ generator that transforms an enum declaration to C++. First all declared enum types are collected from the desugared OIL specification (line 1), after which a C++ enum class is printed line by line for each enum type (lines 2–9).

The code generators in Python also support the use of static analysis results and configurability. Static analysis results and properties are stored by dynamically adding new fields to the classes that represent terms. This information can then be accessed directly when needed. Configuration options are stored globally, which can be directly accessed from anywhere in the translation.

8.5 Evaluation

To evaluate the productivity of implementing dynamic semantics, we look at a single evaluation point: the implementation of code generation. More specifically, we look at the mCRL2 and C++ generators that are available in the Spoofax and Python implementation.

Question. Does it cost less code volume to define code generation for OIL in Spoofax compared to in Python?

Method. We measure the SLOC of the code generators used to transform a desugared and analyzed OIL specification to mCRL2 and to C++. Any SLOC called by the code generators, such as helper functions, are counted too.

Results. Table 4 shows the SLOC used to implement the mCRL2 and the C++ generator in both Stratego and Python. Any SLOC that are used for more purposes than one code generator are captured in the "Reused" row.

Table 4 SLOC for the implementation of the mCRL2 and C++ generator in both the Spoofax and Python implementation of OIL

Artifact	Spoofax	Python
mCRL2 generator	689	531
C++ generator	705	1321
Reused	508	369
Total (Without reused)	1394	1852
Total (All)	1902	2221

Because the exact differences in functionality (of generated code) between the two implementations and what SLOC attributes to these differences is very complex to measure, we decided to measure the SLOC of the code generators in full. This complexity is due to multiple factors. One is that structure of the code generators is very different: the code generators in Spoofax are split up into multiple transformations between IRs, while the code generators in Python do a direct translation from a (desugared) OIL specification to the target. Another is that code generators do not almost only differ in syntactic OIL constructs they support, as is the case for concrete syntax and static semantics, but also what they support in the functionality of the generated code, which is much more difficult to compare accurately.

The Spoofax implementation of the mCRL2 generator consists mainly of Stratego code from the OIL2mCRL2 project (689 SLOC). This code generator also uses the mCRL2 project (373 SDF3 SLOC), but since this project already existed outside the scope of our project, we do not include the SLOC measurements of this project in our results. The Spoofax implementation of the C++ generator consists mainly of Stratego code from the OIL2GPL and GPL projects (389+151 SLOC) and SDF3 code for defining the grammar of the semantic IR from the GPL project (165 SLOC). The reused code in the Spoofax implementation consists mainly of SDF3 code for defining the grammar of the semantic IR from the SEM project (164 SLOC) and Stratego code for various helper transformations used by more than one code generator. The mCRL2 and C++ generators in the Python implementation are both implemented in separate files. The shared Python code consists of helper functions that are used for both code generators.

Analysis. The Spoofax mCRL2 generator uses a factor of 1.3 SLOC compared to the Python mCRL2 generator. The Spoofax C++ generator uses a factor of 0.39 SLOC compared to the Python C++ generator. A big reason for the difference in SLOC ratio of the two code generators is the difference in maturity. The Python mCRL2 generator and the Stratego C++ generator are prototypes that only implement basic code generation. The Stratego mCRL2 generator and the Python C++ generator have more functionality and have been maintained extensively compared to their prototype counterpart.

Diving deeper into the code generators shows that a big difference between the Stratego and the Python implementation is in the use of IRs. In the Spoofax implementation, the code generator consists mainly of model-to-model transformations. The actual target syntax is created using the pretty printer that is automatically generated from the syntax in case of the mCRL2 generator, and using a model-to-text transformation from GPL in case of the C++ generator (see Fig. 34). In Python, no IRs are used: the target code is printed line by line while using the (desugared) Minidom AST of the OIL specification to collect information (see Fig. 40).

The use of IRs does come with the overhead of defining the IRs. Both IRs have been defined by means of SDF3 grammar; the semantic IR uses 164 SLOC and the GPL IR uses 165 SLOC. These could have been implemented with fewer SLOC if implemented with signatures in Stratego like with the normalized and desugared IR, as only these signatures are necessary for the transformation. With 52 constructors for the semantic IR and 44 constructors for the GPL IR, the signatures could be implemented with 1 SLOC per constructor. The difference in SLOC compared to SDF3 is mainly due to the syntax needing lexical elements, needing priority definitions and the definition of some constructors in SDF3 being spread over multiple lines for better pretty printing. However, using SDF3 for this does give the benefit of having a readable syntax and the automatic generation of a pretty printer, which has been proven useful when debugging transformations.

Concerning the comparison of using Stratego over Python for the actual transformation, the same benefits and downsides as for desugaring transformations hold here. Stratego's use of ATerm as data format and its core support for pattern matching helps writing transformations in a concise way, but its not possible to directly access the parent of a term and the immutability of ASTs makes it impossible to create (global) references to parts of an AST. Given the structure of the transformations for the code generators, where the input AST is only used to collect information and the target AST is built up from scratch, these downsides have less of an effect here compared to desugaring, where information collection and transformation is done in the same AST. Specifically for code generators, there is another small downside of using Stratego over Python in the form of the use of properties. In the Python implementation, due to the mutability of the AST, properties can be added and extracted with a single operation. In the Spoofax implementation, as shown in Sect. 8.2, Fig. 37, multiple operations are necessary to retrieve the declaration of an enum type.

The reused Python code mainly consists of general helpers for information collection, AST traversal and code generation. The reused Stratego code mainly consists of transformations to and on the semantic IR, which include calculation steps necessary for the semantics of OIL, such as computing transition pre/postconditions. In the Python implementation, these calculation steps are done separately in both code generators, resulting in some code duplication. This code duplication could have been avoided by creating more shared helper functions, reducing the total amount of SLOC.

The modularity that comes with splitting up the code generators in the Spoofax implementation into multiple projects, as shown in Fig. 32, also comes with a cost in SLOC. To configure the projects such that they extend each other, 31 SLOC is used. Since the code generators in the Python implementation are defined in a single file, no SLOC is needed for anything similar.

Conclusion. To implement an mCRL2 and a C++ generator, the Spoofax implementation uses a factor of 0.86 SLOC compared to the Python implementation (a factor of 0.75 when not counting reused SLOC). With the differences in functionality between the Spoofax and Python code generators in mind, we cannot not draw a conclusion on the definition of code generators.

Discussion. The use of annotated information was one of the things that was most difficult to get working correctly. Because the information is stored on scope graph nodes instead of the terms itself, the information is not easily retrievable. With the NaBL2 interface for Stratego, it was not possible to extract the scope graph nodes that belong to a term directly from this term. Some ideas for NaBL2 properties, such as whether a variable reference refers to one declared in an OIL specification or an operation parameter, were never implemented due to this. The idea to put the scope graph node inside a type as described in Sect. 8.2 is actually more of a workaround, as the type of a term is easily retrievable with the NaBL2 interface. We are not sure whether this is an issue of the NaBL2 interface or of the lack of documentation on it. In Statix, the successor of NaBL2, properties are directly associated with terms instead of scope graph nodes, which alleviates this issue.

A benefit of the model-to-model approach with IRs is reusability of transformations. A good example of this is the definition of C++ methods. In the Stratego implementation, C++ methods are created by defining GPL methods first. Only a single transformation rule needs to be defined to translate a GPL method to a C++ method. In the Python implementation, the syntactical details of each method are repeated every time a new method is defined.

Another benefit of using IRs is that it is good for the extensibility of the implementation. Adding a Java generator to the Stratego implementation only requires a translation from GPL to Java in the GPL IR project, which reuses the transformation to the semantic IR and the transformation from the semantic IR to the GPL IR (389 out of 540 SLOC of the C++ generator in Table 4). In the Python implementation, a new transformation from a desugared OIL specification would need to be defined. This is assuming that the GPL IR is capable of representing all Java constructs that are necessary for the resulting output. If that is not the case, adjustments need to be made to the GPL IR and any transformation to and on it.

9 Evaluation

In this section, we summarize the main findings for our research question and we discuss their threats to validity.

9.1 Summary

RQ: *How does the productivity of implementing an industrial language in Spoofax compare to the productivity when using a GPL and available libraries?*

To answer this, we have measured and compared the code volume used to implement language engineering artifacts in the Spoofax and Python implementations of OIL. Both evaluated implementations are complete, in the sense that all five desired OIL features as described in Sect. 3.3 are realized, except for OF1 (Multiple Syntaxes). OF1 is not implemented in the Python implementation, which is why we only compared SLOC for OILXML. For concrete syntax, abstract syntax, and static semantics we compared artifacts produced by both implementations with similar functionality. For dynamic semantics, we could not make a clear comparison between the Spoofax and the Python implementation due to the large difference of maturity of the mCRL2 and C++ generators of the two implementations.

For concrete syntax, the Spoofax implementation uses a factor of 0.29 SLOC compared to the Python implementation. This difference is mainly caused by the fact that most concrete syntax artifacts are automatically generated from the SDF3 grammar definition in Spoofax, while in Python they are manually implemented, though reusable for other XML-based languages. When not counting these reusable parts, the Spoofax implementation uses a factor of 1.81 SLOC instead compared to the Python implementation.

For abstract syntax we considered AST representations and desugaring transformations. Since the ASTs are represented very differently in both implementations, we could not derive an insight. Comparing the code volume for desugaring transformations, we found that the Spoofax implementation using Stratego uses a factor of 0.56 SLOC compared to the Python implementation (a factor of 0.82 SLOC when not counting reused SLOC). The difference is mainly due to Stratego's core support for pattern matching on ASTs, although the immutability of ASTs in Spoofax can be inconvenient when specifying transformations.

For static semantics the Spoofax implementation uses a factor of 0.49 SLOC compared to the Python implementation. Especially NaBL2's declarative nature, where name binding and typing are defined by means of scope graph and constraint generation rules, helps with keeping the implementation concise.

In summary, for concrete syntax, desugaring transformations and static semantics, the code volume used in Spoofax was about a factor 0.5 or less compared to Python. This is mainly due to the availability of meta-DSLs that are tailored to implementing language development aspects and to generating editor services. When not counting reused SLOC, the results are somewhat more favorable for the Python implementation. Since the comparison is on two implementations covering similar functionality, the results are an indication that it is more productive to implement OIL in Spoofax than in Python.

9.2 Threats to validity

We discuss threats to our study's construct, internal, and external validity. We discuss using code volume as proxy for productivity both as construct- and internal validity.

9.2.1 Construct validity

Construct validity concerns to which extent our code volume measurements actually assess productivity. As threats to construct validity, we discuss using code volume per artifact as proxy for productivity and bias in artifact selection.

9.2.2 Code volume per artifact as proxy for productivity

Using code volume per artifact as a proxy for measuring productivity is a controversial measure [51–54] and a threat to construct validity. Especially for measuring absolute productivity the measure is controversial, as many other factors could have influenced the effort it took to create an implementation. For example, developers can spend the majority of their time on program comprehension and only a small portion on writing code [60]. In general, to mitigate the threat of using code volume per artifact as a proxy, we use the code volume measurements to compare two implementations, not to derive absolute productivity numbers. Second, both implementations already existed before the evaluation, which counters the threat that one implementation could have been optimized in terms of code volume to get better evaluation results. Third, in each evaluation we aim to compare parts of both implementations that cover the same functionality.

A threat that remains is that implementing a DSL is not just about writing lines of code, but also about the time needed to understand how to do so with the implementation language(s) available. The average time per SLOC is influenced by the experience of the developer and the language that is used, e.g., Python is more commonly known than Spoofax and its meta-DSLs. Also, earlier experience with language engineering or compiler construction is beneficial. From our experience, especially NaBL2 requires considerable time to learn. The Master students that contributed to the project seemed to pick up Stratego rather quickly.

We will now discuss using code volume as proxy for productivity in more detail for specific language aspect evaluations. In the concrete syntax evaluation, the original Spoofax and Python implementations did not cover the exact same syntactic languages. For example, this is due to the Spoofax implementation still containing some language constructs that have been removed from the Python implementation. To increase the fairness of our comparison, we have subtracted the lines of code for syntactical elements that are not present in the other implementation. In the Spoofax implementation, this was 31 out of 396 SLOC (7.8%). In the Python implementation, this was 46 out of 1306 SLOC (3.5%). Compared to the productivity comparison outcomes, these differences are inconsequential.

From the abstract syntax evaluation we take Fig. 26 as an example. The desugaring rule *auto-value* in the Python implementation could have been implemented using list comprehension to do multiple steps on the same line of code. This would reduce the lines of code but would also make the code more complex to understand. These threats are mitigated by the fact that both implementations have been created without the goal of evaluating them, let alone optimizing the lines of code, rather than with the goal of being correct and well maintainable.

9.2.3 Bias in artifact selection

In our evaluations, we measure code volume for a selection of artifacts; our selection of artifacts could be biased. This raises the question how representative the selected artifacts are for the whole implementations and thereby is a threat to construct validity. Since for every language aspect the selected artifacts cover almost the whole implementation, we think this threat is negligible.

Next to the implementation of OIL in Spoofax's meta-DSLs and in Python, an implementation of a DSL also contains other code for, e.g., configuration and the build system. We have not included these in the measurements, which could make our measurements less representative for the whole implementations. From our experience, the code spent on configuration and build specification is so little that we do not expect the outcomes of our study to be different if they were included.

Both implementations contain code that is specific to some artifact and code that is reused for multiple artifacts. Some reusable code can even be used beyond OIL, which is especially the case for the implementation of concrete syntax in Python. Since reusability of code impacts productivity, we measure reused code separately and discuss how reusable the code is. When comparing both implementations, we compare both with and without reusable code.

9.2.4 Internal validity

Internal validity concerns to which extent our measurements actually represent the effect on productivity, and cannot be caused by other factors. For internal validity, we discuss using code volume per artifact as proxy for productivity, design decisions, confirmation bias, and experience of developers.

9.2.5 Code volume per artifact as proxy for productivity

In the static semantics evaluation, not much SLOC has been measured for well-formedness compared to name binding and typing, because not many well-formedness constraints were implemented in Spoofax and most of those that are, do not correspond well with constraints in the Python implementation. One of the main reasons for this is that well-formedness was not a high priority during the development of OIL in Spoofax. Therefore, we cannot give a strong indication regarding the productivity of implementing well-formedness.

9.2.6 Interdependence of implementations

Both implementations were not created entirely independently from each other. The Python implementation was already well maintained when we started with the Spoofax implementation. When developing the Spoofax implementation, the Python implementation was used to determine what should be implemented, for instance, which desugaring transformations are necessary and what the code resulting from the C++ code generator should look like. However, the Python implementation was not used to determine *how* things should be implemented in the Spoofax implementation. The meta-DSLs of Spoofax differ from Python so much that there is no clear translation from Python to a meta-DSL, or vice versa. Therefore, we believe that the SLOC measured in one implementation are independent of the SLOC measured in the other implementation.

9.2.7 Design decisions

During the implementation of a DSL, several design decisions are made that influence the implementation. Therefore, particular design decisions can have influenced the outcomes of our study. We have countered this threat by taking two implementations of the same language that are realized independent of our evaluation, i.e., the implementations already existed before this evaluation was started.

The question remains whether the conclusions could have been different given totally different design decisions. For the Spoofax implementation, we think different design decisions would not lead to very different conclusions, as Spoofax and its meta-DSLs steer design decisions, leaving little design decisions to the language engineer. Also, several design decisions that were made in the Spoofax implementation, such as using language composition and many modules for code organization, came with overhead increasing the counted SLOC. For the Python implementation, we think many design decisions could have been made very different, which can steer the implementation to use more or less lines of code, which is a threat to internal validity. Given the nature of our study, where we focus on a complex industrial case, we think this threat is justified. Although all the services of Spoofax could be re-implemented using Python and offered as reusable code, that is not what typically happens as it would be over-design from the perspective of developing a single language.

9.2.8 Confirmation bias

Some of the authors have contributed to the Spoofax and Python implementations of OIL, which raises a concern regarding confirmation bias. We have mitigated the risk of confirmation bias in the following ways, which prevents the possibility for authors to, during the study, change the implementations or steer evidence in a way that supports prior beliefs. First, we have chosen a fixed version of the Spoofax and Python implementations of OIL from a moment before the SLOC measurements started. Second, while code measurements are conducted by a single author, at least one other author has checked these measurements. Third, the authors involved in the implementations of OIL had many discussions to ensure that code measurements cover those parts of the implementation to make comparisons as fair as possible.

9.2.9 Experience

Not all developers that worked on the Spoofax implementation were familiar with Spoofax and its meta-DSLs. Therefore, it could be that the meta-DSLs were not used optimally, and code is unnecessarily large at some points in the implementation. We do not expect this to have large impact on the outcomes of our study. For the Python implementation this is not much of an issue as it is a language (paradigm) that the developers were more experienced with.

9.2.10 External validity

External validity concerns to which extent our findings are generalizable to other language workbenches, comparison to other GPLs, other DSLs, and other contexts. Our study focuses on a particular language workbench (Spoofax), a comparison with a particular GPL (Python), a particular DSL (OIL), and a particular context (the industrial context of Canon Production Printing). Therefore, it is unclear to what extent our findings also hold for other language workbenches, comparison to other GPLs, other DSL cases, and other contexts, as a specific case study is not easy to generalize. OIL's implementations in Spoofax and Python cannot be published due to confidentiality reasons, which hinders the reproducibility of our study.

9.2.11 Generalizability of Python

The Python implementation heavily relies on object-oriented programming and the availability of, e.g., parsing libraries. These aspects are not uncommon for other GPLs and therefore we expect that our findings can be similar for comparisons to other GPLs. Features that are more specific to Python, such as list comprehension, are rarely used in the Python implementation.

9.2.12 Generalizability of OIL

We do think our case is representative of industrial DSL development because OIL is a complex DSL with requirements specific to the industrial context. Still, OIL has specific characteristics that could be very different from other DSLs. Many DSLs only have one syntax, while OF1 required the support for multiple syntaxes. Also, OIL is dependent on another language, IDL, following OF4, while DSLs are often self-contained. On the other hand, relating to OF3, OIL has rather simple typing and name binding rules. We think that desugaring transformations (OF2) and code generation (OF5) are rather common for DSLs, though the structure of the transformations and generators may differ, and some DSLs may be interpreted instead.

10 Discussion

While our evaluation in the previous section is based on conclusions drawn from our quantitative analyses, in this section we discuss aspects of our case study that are of a qualitative nature. First, we discuss the strengths and weaknesses of Spoofax that we have experienced. Second, we list the lessons learned from our study. Finally, we suggest an engineering agenda for Spoofax. In the engineering agenda for Spoofax, we also discuss if and how the weaknesses of Spoofax we have encountered are improved upon in the next version of Spoofax.

10.1 Spoofax's strengths

We list several aspects that worked out well in using Spoofax.

10.1.1 Meta-languages suitable for OIL

The meta-languages that are used (SDF3, NaBL2 and Stratego) all offered sufficient support for implementing OIL's concepts. SDF3 was sufficient for the implementation of OILXML's grammar and enabled rapid prototyping of OILDSL. The name binding and typing features of OIL and IDL could be specified in NaBL2 using the scope graph model. OIL's transformations and code generators could be implemented using Stratego.

10.1.2 Modular language implementation

All meta-languages supported modular language implementation in the sense that implementations could be split up in modules that could be composed or reused. This was beneficial to the Spoofax implementation in many ways. For example, reusing SDF3 modules for shared expression grammar prevented the need to define duplicate grammar rules for the four input languages (see Sect. 5.2). Stratego allows modular and composable definitions of transformations. In particular, Stratego enabled us to implement additional AST schemas and the resilient staging framework, which helped in creating a modular transformation architecture. Lastly, there is little overhead in creating new (composed) languages, which enabled us to easily add an extra language (IDL-OIL-TEST-DSL) specific for testing scenarios of multiple IDL and OIL specifications in isolation.

10.1.3 IDE support

Spoofax derives several editor services automatically for language implementations: parsing, AST inspection, syntax highlighting, syntax error recovery, showing type information, reference resolution, execution of analysis and transformations on file changes, execution of transformations on user request, and marking errors on the specifications. This made it feasible for us to realize an IDE for OIL. The ability to offer a DSL with a user-friendly IDE is beneficial for the adoption of DSLs in an industrial environment such as Canon Production Printing.

10.1.4 Language testing

SPT was useful for testing the implementations of IDL and OIL and to maintain implementation correctness while evolving the languages. SPT supports testing of several (isolated) aspects of the languages such as parsing, name resolution, and typing, as well as end-to-end tests for testing the translations to C++ and mCRL2. Testing helps in obtaining a reliable language implementation and validation during language evolution. For example, when adding or changing functionality to OIL, tests help to ensure other functionality is maintained.

10.1.5 Integration support

Spoofax contains three features for integrating a language implementation within a software ecosystem. First, Stratego offers a Java API which makes it possible to manually implement a transformation rule in the general-purpose language Java, which also enables integration of external tools. This Java API has been used to integrate the SAT solver Z3 for static analysis to optimize generated C++ code [47]. This could also enable automated integration with the mCRL2 toolset, i.e., by automatically calling mCRL2 in a transformation. Second, Spoofax languages can be built outside Eclipse using the Maven or Gradle build systems. This should make it possible to integrate OIL in larger software builds such as continuous integration (CI) or production builds, which is relevant for software development at, e.g., Canon Production Printing. Third, Spoofax offers a Java API (named Spoofax Core) which enables to integrate parts of a Spoofax language implementation such as the parser or transformations within the Java ecosystem. Potentially, these features in combination can enable the integration into an existing industrial software ecosystem.

10.2 Spoofax's weaknesses

We list several aspects that did not work out well in using Spoofax.

10.2.1 Limited portability

Portability concerns to what extent software can be used in different environments. Spoofax currently only provides full support in Eclipse as the IDE for language development and limited support for IntelliJ IDEA. This lack of portability limits the practical applicability opportunities of the language workbench. For example, at Canon Production Printing, software engineers mainly use the Visual Studio IDE, which is currently not supported by Spoofax. Although Spoofax does support integrating parts of a language implementation outside Eclipse using the Java API, the meta-DSLs are not available as independent libraries, hindering integration with other tooling.

10.2.2 Building and runtime performance

The language development experience in Spoofax is hindered by long build times and long response times after changes, sometimes blocking you for minutes. Although it is workable, it does not conform to the response times expected from interactive systems. The editing experience is non-concurrent, e.g., while a build is busy and one changes a file, the build first has to finish before the changed file gets reanalyzed. If a project consists of multiple subprojects, all subprojects have to be built manually one by one in the correct order, because automatic derivation of the correct build order is lacking. This especially becomes cumbersome in a project such as OIL that consists of 14 subprojects, which together take about 16 min to build on a company provided laptop. When changes are made, the projects that are affected by the changes need to be rebuilt. Especially in an industrial context this is a problem, as costly time of engineers is spent on building projects rather than actual development.

10.2.3 Cross-language static analysis

Spoofax with NaBL2 does not offer native support for merging scope graphs of languages to realize language composition on the static semantics level. Conceptually, language composition on the static semantics level using scope graphs boils down to merging the root node of two languages' scope graphs. In practice, this required a workaround by merging the language definitions of IDL and OIL in one language project which accepts both IDL and OIL specifications (see Sect. 7.2). This is a workaround that could be resolved if Spoofax would offer coupling separately-defined languages by merging their scope graphs. The languages could then live next to each other, with their own file extensions, and only interact on scope graphs during static analysis.

10.2.4 Lack of static checking and debugging in NaBL2 and Stratego

The language development experience in NaBL2 and Stratego sometimes was hindered by the limited static checking of specifications in the meta-DSLs. As a result, it often occurs that errors made in a specification are only encountered during execution. For example, a Stratego strategy can fail on getting an incompatible type of term as input which could have been statically checked if strategies were typed. Also, no interactive debugging support for transformations is available. When transformations fail, stack traces are reported without references to the source code with line numbers. This is problematic in an industrial context as it makes engineers spend more time on debugging.

10.2.5 Using static analysis in transformations

Using the NaBL2 analysis results in transformations is cumbersome because low-level querying of the scope graph is required for general operations such as finding a declaration given a reference (see Sect. 8.2). The API is also sparsely documented, which makes it unclear how the API should be used. Spoofax could improve here by offering abstractions for common static analysis querying patterns.

10.2.6 Language evolution and refactoring

Evolving a language implementation in Spoofax can lead to cumbersome situations. For example, when IDL and OIL change, all specifications written in IDL and OIL have to be migrated manually. If the signature of a term changes, many Stratego transformations may need to be migrated as well. This has occurred in practice, for instance, when area type "scope" was renamed to "zone". Applying the change of a name throughout the implementation involves intensive searching and replacing. Spoofax could be improved by adding support for cross-project and cross-meta-DSL refactorings in language definitions, similar to how modern IDEs support this.

10.2.7 Fine-grained testing

SPT mostly supports end-to-end testing of language implementations, whereas it was often desired to test individual parts of the implementation in a more fine-grained manner. For example, it was only possible to test desugaring transformations with SPT by defining tests that, given an OIL specification, generate the normalized IR, apply the desugaring transformation on it, and then transform it back to the original syntax. The success of this test does not only depend on the desugaring transformation, but also on the transformations between the textual OIL specification and the normalized IR. It would be useful if, in SPT, one could write a test for a particular transformation rule or strategy for an input directly written as ATerm, essentially unit testing a small part of a transformation.

10.2.8 Editor actions for configurable code generators

As discussed in Sect. 8.3, some code generators have a number of configuration options. A user can pick values for these configuration options when selecting a code generator in editor action menus, which are defined using ESV (Spoofax's meta-DSL for defining editor services, see Sect. 2.6). However, it is not possible in ESV to reuse (sub)menus; every (sub)menu and menu item must be defined explicitly. When adding a new configuration option with *n* possible values, *n* times more editor actions need to be defined, which makes the size of the ESV file exponential in the number of configuration options.

10.3 Lessons learned

We list our most important lessons learned from implementing OIL in the industrial context of Canon Production Printing both using Python and using Spoofax 2:

1. The meta-DSLs in Spoofax are just like DSLs limited to a certain domain, and it is not unheard of that we end up at the edges of this domain. For us, the meta-DSLs in Spoofax have been sufficient in the industrial context. Except for a few practical workarounds, we have experienced no limitations in implementing concrete syntax (with SDF3), abstract syntax (with Stratego), static semantics (with NaBL2 for typing and name binding and with Stratego for well-formedness checking), and dynamic semantics (with Stratego).
2. The biggest limitations of Spoofax 2 are not in the functional aspects of meta-DSLs, but in their non-functional characteristics, e.g., slow build and response times, limited documentation, limited portability, and limited static checking of meta-DSL specifications.
3. Choosing XML and Python is a viable engineering choice in the absence of a language workbench. XML is a good choice for an effective implementation of concrete syntax for a DSL if dependence on external tools is undesired. Therefore, this is a simple alternative to using a language workbench with a penalty of roughly twice the code size and half of the editor features, as well as a penalty in the user-friendliness of the language.
4. A main benefit of DSLs is the multiplicative factor: from a single specification in a DSL, multiple backends can be targeted or multiple artifacts can be generated. This multiplicative factor is essential for the effectiveness of meta-DSLs used to implement DSLs: a single specification in a meta-DSL can generate multiple language processing artifacts and editor services. For instance, from an SDF3 grammar, not only a parser is generated, but also an AST schema, a pretty printer, origin tracking and editor services.
5. Separate meta-DSLs for separate language implementation aspects lead to a clear separation of concerns, making it effective to define and maintain language aspects within those concerns. From our experience, the fundamental design decision of Spoofax to have clearly separated meta-DSLs seems to be working well.
6. Specifications written in Spoofax's meta-DSLs can have high reusability and extensibility, by decomposition into modules, but this can come with a considerable cost in terms of code to compose the modules. However, since this code almost only consists of declaring and importing

modules, we recommend to use Spoofax's meta-DSLs in a modular way.

10.4 Spoofax engineering agenda

Based on our experiences from implementing OIL with Spoofax 2, we suggest the following improvements to make on the language workbench to increase its adoptability in industry. We have presented these items to the Spoofax development team and incorporated their responses with respect to if and how these items have improved upon in Spoofax 3.

10.4.1 Portability

By making Spoofax available to more IDEs, more developers could make use of it in their IDE of choice. When companies have a policy on which IDEs engineers should use, not supporting such IDEs can block Spoofax from being adopted. Potentially, adding Language Server Protocol (LSP) support can help in improving Spoofax's portability; in principle the support needs to be implemented once but will make Spoofax available to all IDEs that support LSP. Although Spoofax 3 is not more portable out of the box (it supports Eclipse, Gradle, and a command line interface), it features a fundamentally different architecture than Spoofax 2. By supporting static rather than dynamic loading of languages, it is easier to extend Spoofax 3 with support for other IDEs or LSP. Custom language integrations are also easier to make, as languages can be packaged as Java libraries. In addition to languages developed with Spoofax, it would also be useful to offer the meta-DSLs as libraries, as that would ease integration with other tooling and allows the meta-DSLs to reach wider audiences.

10.4.2 Language build system

Several improvements can be made to the language build system provided by Spoofax to improve the development experience: improving build times (e.g., by further incrementalizing builds), automatically building a project that consists of multiple subprojects in the right order based on dependencies, and automatically checking whether exports and imports of files between projects are valid such that errors are detected early and do not require trial and error to debug. Spoofax 3 improves on all these aspects with the introduction of the PIE (Pipelines for Interactive Environments) [29] build system which features fully incremental builds for implementing both Spoofax 3 itself as well as languages developed with Spoofax 3. This is also relevant to debugging Stratego 2 code: with quick enough compilation round-trip, print debugging becomes much more viable.

10.4.3 Runtime performance

Improving the response times after changes in Spoofax would improve the development experience, such that less time during development is spent on waiting. In Spoofax in Eclipse, concurrent editor actions would improve the development experience; currently, e.g., when a build is busy, changes to other files are only picked up after the build finishes. Spoofax 3 with incrementalization using PIE improves runtime performance of both language builds as well as responsiveness of interactions in the IDE. With the introduction of PIE, runtime performance is not better for all implementation aspects as, e.g., the same SDF3 parser generation is used which itself is not incrementalized.

10.4.4 Cross-language static analysis

Spoofax could be improved by supporting cross-language static analysis by making it possible to merge the scope graphs of two separately defined languages, as also described by Zwaan [61]. In Spoofax 2 this was deemed virtually impossible to implement. Thanks to the new architecture of Spoofax 3, it supports the implementation of cross-language static analysis, which should support the OIL and IDL case. Cross-meta-language static analysis was one of the goals of Zwaan [61], but have not yet been materialized.

10.4.5 Static checking in meta-DSLs

Improved static checking in Spoofax's meta-DSLs would enhance the language development experience by reducing the need for trial and error. The next version of Spoofax partly improves on these aspects, in the meta-DSLs Stratego 2 and Statix (successor to NaBL2). Stratego 2 introduces gradual typing [23] and Statix comes with static checks on its specifications.

10.4.6 Stratego debugging

Spoofax only supports debugging of Stratego transformations by adding debug transformation rules that print information to the console. It would be beneficial for development with Stratego to be able to step through a transformation interactively, while showing the values of local variables and the term that the transformation is applied on. Spoofax 3 and Stratego 2 do not yet support debugging of transformations.

10.4.7 Integrating static analysis with transformations

An improved API for using static analysis results in transformations can make transformation definitions more simple. In NaBL2's successor, Statix [25], some issues are alleviated already. For instance, in Statix properties are defined on terms directly instead of on scope graph nodes, which makes querying them straightforward.

10.4.8 Documentation

Improved documentation will help engineers new to Spoofax to learn and adopt the tool, without having to learn from experiences from others or by experimentation.

10.4.9 Unit testing Stratego

It would be desired to have core support for unit-testing Stratego transformation rules in SPT, by supplying an input term, a transformation, and an expected result term. Especially for large and modular transformation architectures, the restriction of only testing end-to-end transformations makes it difficult and cumbersome to test individual parts of the transformation architecture.

10.4.10 ESV-Stratego integration

A better integration between ESV and Stratego could make the definition of editors services more simple. For example, supporting parameterized Stratego transformations in ESV would avoid redundant definitions. In the current state of ESV, the size of an ESV specification grows exponentially in the number of configuration options available for an end-to-end transformation.

11 Related work

We discuss related work on evaluating language workbenches (and tools to develop DSLs in general) and other process languages such as OIL.

11.1 Language workbench evaluation

We first discuss Spoofax and then other language workbenches.

Most research on Spoofax focuses on the language workbench's fundamentals, with artificial languages as examples. An exception of this is Visser's case study on the development of WebDSL [62], which discusses language design and implementation for a DSL in the domain of web programming. The paper highlights the DSL development process and how the different aspects of this process can be covered by the meta-DSLs of Spoofax. This study uses Spoofax version 1, the predecessor of the Spoofax version used in our study. Several discussion sections cover DSL engineering evaluation criteria focusing on the process and the language that is produced, not on the tools for developing the language (SDF + Stratego), language engineering paradigms, or language engineering challenges. Therefore, this work does not evaluate Spoofax itself or how it compares to not using a language workbench. Hamey and Goldrei [63] reported on their experiences of using SDF and Stratego compared to using traditional techniques. They found that the Stratego toolset enabled easy implementation with opportunity of enhancing the language and improving performance of generated code, compared to the implementation using traditional techniques.

Canon Production Printing uses modeling languages across various engineering disciplines [64]. Schindler et al. describe how the company envisions the use of models during the complete life cycle of printers to address the challenges of efficiently performing continuous innovation with sustainable quality. The MPS language workbench is selected as one of the core technologies to develop custom DSLs that can interconnect the models from different engineering disciplines and tools. The authors find that using modeling approaches has advantages: users only have to learn a single tool, multiple models can be generated from a single tool, and one point of maintenance is needed instead of multiple. They also encountered challenges in using MPS: steep learning curve, lack of full-fledged DSL models in MPS for commodity languages such as C++, existing parsers or grammars are not immediately reusable, and performance can be undesirably low.

Voelter et al. [12] report on their experiences on using MPS for the development of mbeddr, a large set of languages and extensions of the C language that targets embedded software development. This work is, to our knowledge, the largest evaluation of a language workbench, spanning a case that involved around 10 person years of development effort in an industrial setting. Whereas our work is centered around evaluating productivity, the paper by Voelter et al. is centered around five topics concerning the use of MPS: language modularity, notational freedom and projectional editing, mechanisms for managing complexity, performance and scalability issues, and consequences for the development process. The authors draw generally positive conclusions and indicate various places for improvement as well.

Broccia et al. [65] state that although the quantitative aspects of language workbenches are often discussed in literature (e.g., the evaluations and comparisons by Erdweg et al. [5]), the evaluation of comprehensibility of the meta-languages used in language workbenches is typically neglected. The authors evaluate the Neverlang [66] language workbench on four aspects. First, the comprehensibility of programs in Neverlang in terms of users' effectiveness and efficiency in code comprehension tasks. Second, the relationship between comprehensibility and users' working memory capacity. Third, to which extent users consider the language workbench acceptable in terms of perceived ease of use, usefulness, and intention to use. Fourth, how comprehensibility relates to the degree of acceptance of the language. The study suggests that users' working memory capacity may be related to the ability to comprehend Neverlang programs. Effectiveness and efficiency do not appear to be related to an increase of users' acceptance variables. We believe more studies like these can be useful for getting a better understanding of how language workbenches are perceived and what influences their adoption.

Klint et al. [67] found that using DSL tools (ANTLR, OMeta, Microsoft "M") improve the maintainability of language implementations by comparing several implementations of the same DSL both with and without the use of DSL tools; the implementations without DSL tools use GPLs (Java, JavaScript, C#). The evaluation considers parsing, static analysis, and transformation. The results suggest that DSL tools increase maintainability of DSL implementation compared to using GPLs. The work is similar to our work by comparing implementations of a DSL using GPLs to implementations using tools specific for DSL development. The work by Klint et al. differs from our work in the sense that they compare six implementations instead of two, the DSL tools do not cover aspects of language engineering such as deriving IDEs, and they focus on maintainability instead of productivity.

Åkesson et al. [68] report on their experiences on the implementation of a Modelica compiler using JastAdd [69] compiler tool. In particular, an aim is to achieve extensibility of the compiler, which led to the choice of using the declarative attribute grammar approach provided by JastAdd. They illustrate how existing design strategies for a Java compiler implemented using JastAdd could be reused for advanced features of the Modelica compiler. The authors show complex semantic rules can be implemented in a compact and modular manner. Given the 9 man-months of development time that was spent on creating the implementation, they find that JastAdd is very well suited for rapid compiler development.

Basten et al. present a language engineering case study on a Rascal implementation of Oberon-0 [70], focusing on how the language can be implemented in a modular way. Oberon-0 consists of four language levels where each succeeding level is implemented as an extension of the preceding level, supported by Rascal's modularity features. The implementation used less than 1500 SLOC which includes the implementation of parsing, name and type analysis, desugaring, transformation, and compilation to C. Additionally, they found directions for improvement for Rascal.

Zarrin et al. [71] report on their experiences of introducing a DSL for material flow analysis using Microsoft DSL tools. Their motivation for using the DSL is to enable domain experts to evolve existing software to fulfill new requirements. The authors report that the DSL tools were mature enough to develop a complete DSL. Drawbacks include redundantly having to define semantics for simulation and code generation, the visualization of the metamodel is difficult to understand, and being limited to graphical notation.

Other implementations exist of DSLs like OIL for the specification of behavior. For instance, the language Dezyne developed by the company Verum[9] can be used to define system behavior and its implementation in Guile[10] includes multiple code generators [72]. The Comma framework[11] contains a collection of languages and tools to define and analyze the signatures and behavior of interfaces and is implemented using Xtext, which also supports many code generators [73]. BPMN[12] is a UML-like graphical language for modeling business processes maintained by the Object Management Group, implemented using MOF (Meta Object Family), with XSD for static semantics and XSLT[13] for dynamic semantics [74]. SystemC[14] is a language for simulating event-driven concurrent processes, defined as a subset of C++ with predefined classes and functions, which makes it possible to reuse much of the already existing analysis and editor services for C++.

12 Conclusions

In this paper, we have presented an industrial case study on language engineering with the Spoofax language workbench. In summary, the contributions of this paper are:

- An evaluation of whether Spoofax's original claims—on making language development, compared to not using a language workbench, more productive—stand when realizing the implementation of a complex industrial language such as OIL.
- Lessons learned on implementing OIL using Spoofax in the industrial context of Canon Production Printing.
- Strengths, weaknesses, and an agenda for future engineering on Spoofax.

We found that Spoofax and its meta-DSLs SDF3, NaBL2 and Stratego were adequate for implementing OIL. It was possible to implement every OIL feature using the meta-DSLs. Several workarounds were needed, such as the interaction between IDL and OIL when it comes to static analysis, but these could still be implemented within Spoofax.

9 https://www.verum.com/.

10 https://www.gnu.org/software/guile/.

11 https://comma.esi.nl/.

12 https://www.bpmn.org/.

13 https://www.w3.org/TR/xslt-30/.

14 https://systemc.org/.

In our evaluation, we found indications that it is more productive to implement a complex DSL with a language workbench compared to not using a language workbench. We did this by comparing the code volume (in SLOC) of two implementations of OIL, one using Spoofax and one using Python, which both already existed before the evaluation. The evaluation shows that the Spoofax implementation used fewer SLOC compared to the Python implementation, while offering more editor features. This is relevant in an industrial setting because it enables to develop a full-featured IDE with less code.

Naturally, our evaluation is not without threats to its validity. The use of SLOC as metric for productivity is contested. For instance, there can be much variance in what a line of code defines. We do feel that the results on code volume per artifact are an indication for higher productivity with Spoofax compared to Python, as the analyses show considerable differences in SLOC for artifacts with the same functionality and because both implementations were created before we had the intent to evaluate them. Since our evaluation is done for a single use case, it is difficult to generalize our findings to other workbenches, languages and contexts. Therefore, we call for more studies on applications of language workbenches in practice. This is relevant because it will help industrial language engineers decide when and how to use language workbenches.

In our study, we have primarily focused on evaluating and comparing productivity. Still, we were able to make several observations for other concerns such as modularity and maintainability of language implementations, both positive and negative. For example, the ability to easily extend SDF3 definitions and Spoofax projects benefits modularity and the ability to generate multiple artifacts from a single source benefits maintainability. On the other hand, the inability to merge scope graphs of different languages hinders modularity and the steep learning curve of NaBL2 hinders maintainability. Since concerns such as the modularity and maintainability of language implementations are important for developing DSLs in industry, we encourage more studies that evaluate language workbenches in detail on dimensions other than productivity.

Although Spoofax was suitable for implementing OIL, we see several areas of improvements. These are mainly in the practical use of the language workbench, such as limited portability, slow build and response times, and limited documentation. For the meta-DSLs, we see the following opportunities for improvement: supporting cross-language static analysis, improving the API for using static analysis results in transformations, supporting unit testing, and improving the integration of Stratego in the definition of editor services. Several of these improvements have been included in the next version of Spoofax.

Based on our study, we provide the following advise.

- *For industrial language engineers:* Use a language workbench for developing DSLs especially if a user-friendly editor for the languages is desired; not doing so leads to “reinventing the wheel”, which can cost considerable effort.
- *For industrial language engineers:* When IDE support is not required, using, e.g., off-the-shelf parser generators and a GPL could be a valid engineering choice for implementing the concrete syntax of a DSL, as the drawbacks of a language workbench may outweigh the benefits.
- *For language workbench developers:* Focus on the practical aspects of language workbenches such as portability, usability, and documentation to improve adoptability.

Acknowledgements This study was started under the guidance of Eelco Visser, who passed away on April 5, 2022. The authors decided to posthumously acknowledge his contributions to this work by making him co-author. We thank Aron Zwaan, Gabriel Konat, Hendrik van Antwerpen, Daniël Pelsmaeker, and Jeff Smits for their feedback on our findings on Spoofax 2, especially regarding how they compare to Spoofax 3. We thank the reviewers for their useful feedback that helped us improve the paper.

Funding Funding was provided by the Top Consortia for Knowledge and Innovation and by Canon Production Printing.

References

1. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. SIGPLAN Not. **35**(6), 26–36 (2000). https://doi.org/10.1145/352029.352035
2. Boersma, M.: Business-Friendly DSLs. Manning (to appear) (2024) 9781617296475
3. van Deursen, A., Heering, J., Klint, P.: Language Prototyping: An Algebraic Specification Approach, volume 5 of AMAST Series in Computing. World Scientific, Singapore (1996). ISBN 978-981-4498-73-9. https://doi.org/10.1142/3163
4. Fowler, M.: Language workbenches: The killer-app for domain specific languages? (2005)
5. Erdweg, S., van der Storm, T., Völter, M., Tratt, L., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G., Molina, P.J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V.A., Visser, E., van der Vlist, K., Wachsmuth, G., van der Woning, J.: Evaluating and comparing language workbenches: Existing results and benchmarks for the future. Comput. Lang. Syst. Struct. **44**, 24–47 (2015). https://doi.org/10.1016/j.cl.2015.08.007

6. Pech, V.: Jetbrains mps: Why modern language workbenches matter. In: Bucchiarone, A., Cicchetti, A., Ciccozzi, F., Pierantonio, A. (eds.), Domain-Specific Languages in Practice: with JetBrains MPS, pp. 1–22. Springer, Berlin (2021). ISBN 978-3-030-73758-0. https://doi.org/10.1007/978-3-030-73758-0_1
7. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: Cook, W.R., Clarke, S., Rinard, M.C., (eds.) Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17–21, 2010, Reno/Tahoe, Nevada, USA, pp. 307–309. ACM, New York (2010). ISBN 978-1-4503-0240-1. https://doi.org/10.1145/1869542.1869625
8. Klint, P., van der Storm, T., Vinju, J.J.: EASY meta-programming with Rascal. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) Generative and Transformational Techniques in Software Engineering III—International Summer School, GTTSE 2009, Braga, Portugal, July 6–11, 2009. Revised Papers, volume 6491 of Lecture Notes in Computer Science, pp. 222–289. Springer, Berlin (2009). ISBN 978-3-642-18022-4. https://doi.org/10.1007/978-3-642-18023-1_6
9. Kats, L.C.L., Visser, E.: The Spoofax language workbench: rules for declarative specification of languages and IDEs. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, pp. 444–463, Reno/Tahoe, Nevada (2010). ACM, New York. ISBN 978-1-4503-0203-6. https://doi.org/10.1145/1869459.1869497
10. Barash, M.: Vision: the next 700 language workbenches. In: Visser, E., Kolovos, D.S., Söderberg, E. (eds.) SLE '21: 14th ACM SIGPLAN International Conference on Software Language Engineering, Chicago, IL, USA, October 17–18, 2021, pp. 16–21. ACM, New York (2021). ISBN 978-1-4503-9111-5. https://doi.org/10.1145/3486608.3486907
11. Van den Brand, M., van Deursen, A., Klint, P., Klusener, S., van der Meulen, E.: Industrial applications of asf+ sdf. In: International Conference on Algebraic Methodology and Software Technology, pp. 9–18. Springer, Berlin (1996)
12. Voelter, M., Kolb, B., Szabó, T., Ratiu, D., van Deursen, A.: Lessons learned from developing mbeddr: a case study in language engineering with MPS. Softw. Syst. Model. **18**(1), 585–630 (2019). https://doi.org/10.1007/s10270-016-0575-4
13. Groenewegen, D.M., Hemel, Z., Kats, L.C.L., Visser, E.: WebDSL: a domain-specific language for dynamic web applications. In: Harris, G.E. (ed.), Companion to the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19–13, 2007, Nashville, TN, USA, pp. 779–780. ACM, New York (2008). ISBN 978-1-60558-220-7. https://doi.org/10.1145/1449814.1449858
14. Groenewegen, D.M., van Chastelet, E., Visser, E.: Evolution of the WebDSL runtime: reliability engineering of the WebDSL web programming language. In: Aguiar, A., Chiba, S., Boix, E.G. (eds.) Programming'20: 4th International Conference on the Art, Science, and Engineering of Programming, Porto, Portugal, March 23–26, 2020, pp. 77–83. ACM, New York (2020). ISBN 978-1-4503-7507-8. https://doi.org/10.1145/3397537.3397553
15. Harkes, D., Visser, E.: Icedust 2: Derived bidirectional relations and calculation strategy composition. In: Müller, P. (ed.) 31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19–23, 2017, Barcelona, Spain, volume 74 of LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. ISBN 978-3-95977-035-4. https://doi.org/10.4230/LIPIcs.ECOOP.2017.14
16. Harkes, D., van Chastelet, E., Visser, E.: Migrating business logic to an incremental computing DSL: a case study. In: Pearce, D., Mayerhofer, T., Steimann, F. (eds.) Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018, pp. 83–96. ACM, New York (2018). ISBN 978-1-4503-6029-6. https://doi.org/10.1145/3276604.3276617
17. Visser, E., Wachsmuth, G., Tolmach, A.P., Néron, P., Vergu, V.A., Passalaqua, A., Konat, G.: A language designer's workbench: A one-stop-shop for implementation and verification of language designs. In: Black, A.P., Krishnamurthi, S., Bruegge, B., Ruskiewicz, J.N. (eds.) Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20–24, 2014, pp. 95–111. ACM, New York (2014). ISBN 978-1-4503-3210-1. https://doi.org/10.1145/2661136.2661149
18. Visser, E.: Syntax Definition for Language Prototyping. Ph.D. thesis, University of Amsterdam, September (1997)
19. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. Sci. Comput. Program. **72**(1–2), 52–70 (2008). https://doi.org/10.1016/j.scico.2007.11.003
20. van Antwerpen, H., Néron, P., Tolmach, A.P., Visser, E., Wachsmuth, G.: A constraint language for static semantic analysis based on scope graphs. In: Erwig, M., Rompf, T. (eds.) Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20–22, 2016, pp. 49–60. ACM, New York (2016). ISBN 978-1-4503-4097-7. https://doi.org/10.1145/2847538.2847543
21. Konat, G.: Language-Parametric Methods for Developing Interactive Programming Systems. Ph.D. thesis, Delft University of Technology, Netherlands (2019)
22. de Souza Amorim, L.E., Visser, E.: Multi-purpose syntax definition with SDF3. In: de Boer, F.S., Cerone, A. (eds.) Software Engineering and Formal Methods—18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14–18, 2020, Proceedings, volume 12310 of Lecture Notes in Computer Science, pp. 1–23. Springer, Berlin (2020). ISBN 978-3-030-58768-0. https://doi.org/10.1007/978-3-030-58768-0_1
23. Smits, J., Visser, E.: Gradually typing strategies. In: Lämmel, R., Tratt, L., de Lara, J. (eds.) Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020, pp. 1–15. ACM, New York (2020). ISBN 978-1-4503-8176-5. https://doi.org/10.1145/3426425.3426928
24. Smits, J., Konat, G., Visser, E.: Constructing hybrid incremental compilers for cross-module extensibility with an internal build system. Program. J. **4**(3), 16 (2020). https://doi.org/10.22152/programming-journal.org/2020/4/16
25. Rouvoet, A., van Antwerpen, H., Poulsen, C.B., Krebbers, R., Visser, E.: Knowing when to ask: sound scheduling of name resolution in type checkers derived from declarative specifications. In: Proceedings of the ACM on Programming Languages, 4 (OOPSLA) (2020). https://doi.org/10.1145/3428248
26. Néron, P., Tolmach, A.P., Visser, E., Wachsmuth, G.: A theory of name resolution. In: Vitek, J. (ed.) Programming Languages and Systems—24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11–18, 2015. Proceedings, volume 9032 of Lecture Notes in Computer Science, pp. 205–231. Springer, Berlin (2015). ISBN 978-3-662-46668-1. https://doi.org/10.1007/978-3-662-46669-8_9
27. Zwaan, A., van Antwerpen, H., Visser, E.: Incremental type-checking for free: using scope graphs to derive incremental type-checkers. In: Proceedings of the ACM on Programming Languages **6**(OOPSLA2), 424–448 (2022). https://doi.org/10.1145/3563303
28. Smits, J., Wachsmuth, G., Visser, E.: Flowspec: a declarative specification language for intra-procedural flow-sensitive data-flow

analysis. J. Comput. Lang. **57**, 100924 (2020). https://doi.org/10.1016/j.cola.2019.100924

29. Konat, G., Steindorfer, M.J., Erdweg, S., Visser, E.: PIE: a domain-specific language for interactive software development pipelines. Program. J. **2**(3), 9 (2018). https://doi.org/10.22152/programming-journal.org/2018/2/9
30. Pelsmaeker, D.A.A., van Antwerpen, H., Poulsen, C.B., Visser, E.: Language-parametric static semantic code completion. In: Proceedings of the ACM on Programming Languages, 6 (OOPSLA), pp. 1–30 (2022). https://doi.org/10.1145/3527329
31. van den Mark, G.J., de Brand, H.A., Jong, P.K., Olivier, P.A.: Efficient annotated terms. Softw. Pract. Exp. **30**(3), 259–291 (2000)
32. Klop, J.W.: Term rewriting systems: From Church-Rosser to Knuth-Bendix and beyond. In: Paterson, M. (ed.) Automata, Languages and Programming, 17th International Colloquium, ICALP90, Warwick University, England, July 16–20, 1990, Proceedings, volume 443 of Lecture Notes in Computer Science, pp. 350–369. Springer, Berlin (1990). ISBN 3-540-52826-1
33. Chomsky, N.: Three models for the description of language. IRE Trans. Inf. Theory **2**(3), 113–124 (1956). https://doi.org/10.1109/TIT.1956.1056813
34. Vollebregt, T., Kats, L.C.L., Visser, E.: Declarative specification of template-based textual editors. In: Sloane, A., Andova, S. (eds.) International Workshop on Language Descriptions, Tools, and Applications, LDTA ’12, Tallinn, Estonia, March 31–April 1, 2012, pp. 1–7. ACM, New York (2012). ISBN 978-1-4503-1536-4. https://doi.org/10.1145/2427048.2427056
35. van den Brand, M.G.J., Scheerder, J., Vinju, J.J., Visser, E.: Disambiguation filters for scannerless generalized LR parsers. In: Horspool, R.N. (ed.) Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings, volume 2304 of Lecture Notes in Computer Science, pp. 143–158. Springer, Berlin (2002). ISBN 3-540-43369-4. https://doi.org/10.1007/3-540-45937-5_12
36. van Deursen, A., Klint, P., Tip, F.: Origin tracking. J. Symb. Comput. **15**(5/6), 523–545 (1993)
37. van Antwerpen, H., Poulsen, C.B., Rouvoet, A., Visser, E.: Scopes as types. Proceedings of the ACM on Programming Languages, 2 (OOPSLA) (2018). https://doi.org/10.1145/3276484
38. Visser, E., Benaissa, Z.-E.-A., Tolmach, A.P.: Building program optimizers with rewriting strategies. In: Felleisen, M., Hudak, P., Queinnec, C. (eds.) Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming, pp. 13–26, Baltimore, Maryland, United States (1998). ACM. https://doi.org/10.1145/289423.289425
39. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**(3), 621–645 (2006)
40. Kats, L.C.L., Vermaas, R., Visser, E.: Testing domain-specific languages. In: Lopes, C.V., Fisher, K. (eds.) Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22–27, 2011, pp. 25–26. ACM, New York (2011) ISBN 978-1-4503-0942-4. https://doi.org/10.1145/2048147.2048160
41. Denkers, J., van Gool, L., Visser, E.: Migrating custom DSL implementations to a language workbench (tool demo). In: Pearce, D., Mayerhofer, T., Steimann, F. (eds.) Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05–06, 2018, pp. 205–209. ACM, New York (2018). ISBN 978-1-4503-6029-6. https://doi.org/10.1145/3276604.3276608
42. Bunte, O., van Gool, L.C.M., Willemse, T.A.C.: Formal verification of OIL component specifications using mCRL2. STTT **24**(3), 441–472 (2022). https://doi.org/10.1007/s10009-022-00658-y
43. Erdweg, S., Giarrusso, P.G., Rendel, T.: Language composition untangled. In: Sloane, A., Andova, S. (eds.) International Workshop on Language Descriptions, Tools, and Applications, LDTA ’12, Tallinn, Estonia, March 31–April 1, 2012, p. 7. ACM, New York (2012). ISBN 978-1-4503-1536-4. https://doi.org/10.1145/2427048.2427055
44. Völter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L.C.L., Visser, E., Wachsmuth, G.: DSL Engineering - Designing, Implementing and Using Domain-Specific Languages. dslbook.org (2013). ISBN 978-1-4812-1858-0
45. Groote, J.F., Mousavi, M.R.: Modeling and Analysis of Communicating Systems. MIT Press, Cambridge (2014). ISBN 9780262321020
46. Frenken, M.: Code generation and model-based testing in context of OIL (2019)
47. Buskens, T. Optimizing the code generator for OIL (2021)
48. Voogd, S.N., Aslam, K., van Gool, L., Theelen, B., Malavolta, I.: Real-time collaborative modeling across language workbenches—a case on Jetbrains MPS and Eclipse Spoofax. In: ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS 2021 Companion, Fukuoka, Japan, October 10–15, 2021, pp. 16–26. IEEE (2021). ISBN 978-1-6654-2484-4. https://doi.org/10.1109/MODELS-C53483.2021.00011
49. van Gool, L.: Formalising interface specifications. Ph.D. thesis, Eindhoven University of Technology (2006)
50. Bunte, O., van Gool, L.C.M., Willemse, T.A.C.: Formal verification of OIL component specifications using mCRL2. In: ter Beek, M.H., Nickovic, D. (eds.) Formal Methods for Industrial Critical Systems—25th International Conference, FMICS 2020, Vienna, Austria, September 2–3, 2020, Proceedings, volume 12327 of Lecture Notes in Computer Science, pp. 231–251. Springer, Berlin (2020). ISBN 978-3-030-58298-2. https://doi.org/10.1007/978-3-030-58298-2_10
51. Nguyen, V., Deeds-Rubin, S., Tan, T., Boehm, B.: A SLOC counting standard. In: Cocomo ii forum, volume 2007, pp. 1–16. Citeseer (2007)
52. Walston, C.E., Felix, C.P.: A method of programming measurement and estimation. IBM Syst. J. **16**(1), 54–73 (1977)
53. Boehm, B.W.: Software engineering economics. IEEE Trans. Softw. Eng. **10**(1), 4–21 (1984)
54. Armour, P.G.: Beware of counting LOC. Commun. ACM **47**(3), 21–24 (2004)
55. Wąsowski, A., Berger, T.: Domain-specific Languages: Effective Modeling, Automation, and Reuse. Springer Nature, Berlin (2023)
56. Fowler, M.: Domain-Specific Languages. Addison Wesley, Boston (2010)
57. Ward, M.P.: Language-Oriented Programming. Software—Concepts and Tools, 15(4) (1994)
58. Bravenboer, M., van Dam, A., Olmos, K., Visser, E.: Program transformation with scoped dynamic rewrite rules. Fund. Inform. **69**(1–2), 123–178 (2006)
59. Bunte, O., Groote, J.F., Keiren, J.J.A., Laveaux, M., Neele, T., de Vink, E.P., Wesselink, W., Wijs, A., Willemse, T.A.C.: The mCRL2 toolset for analysing concurrent systems—improvements in expressivity and usability. In: Vojnar, T., Zhang, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems—25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part II, volume 11428 of Lecture Notes in Computer Science, pp. 21–39. Springer, Berlin (2019). ISBN 978-3-030-17465-1. https://doi.org/10.1007/978-3-030-17465-1_2
60. Minelli, R., Mocci, A., Lanza, M.: I know what you did last summer: an investigation of how developers spend their time. In: De Lucia, A., Bird, C., Oliveto, R. (eds.) Proceedings of the 2015 IEEE

23rd International Conference on Program Comprehension, ICPC 2015, Florence/Firenze, Italy, May 16–24, 2015, pp. 25–35. ACM, New York (2015)
61. Zwaan, A.: Composable type system specification using heterogeneous scope graphs (2021)
62. Visser, E.: WebDSL: a case study in domain-specific language engineering. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, volume 5235 of Lecture Notes in Computer Science, pp. 291–373, Braga, Portugal, (2007). Springer, Berlin. ISBN 978-3-540-88642-6. https://doi.org/10.1007/978-3-540-88643-3_7
63. Hamey, L.G.C., Goldrei, S.: Implementing a domain-specific language using stratego/xt: an experience paper. Electron. Notes Theor. Comput. Sci. **203**(2), 37–51 (2008). https://doi.org/10.1016/j.entcs.2008.03.043
64. Schindler, E., Moneva, H., van Pinxten, J., van Gool, L., van der Meulen, B., Stotz, N., Theelen, B.: JetBrains MPS as core DSL technology for developing professional digital printers. In Antonio Bucchiarone, Antonio Cicchetti, Federico Ciccozzi, and Alfonso Pierantonio, editors, Domain-Specific Languages in Practice: with JetBrains MPS, pp. 53–91. Springer, Berlin (2021). ISBN 978-3-030-73758-0. https://doi.org/10.1007/978-3-030-73758-0_3
65. Broccia, G., Ferrari, A., ter Beek, M.H., Cazzola, W., Favalli, L., Bertolotti, F.: Evaluating a language workbench: from working memory capacity to comprehension to acceptance. In: 31st IEEE/ACM International Conference on Program Comprehension, ICPC 2023, Melbourne, Australia, May 15–16, 2023, pp. 54–58. IEEE (2023). ISBN 979-8-3503-3750-1. https://doi.org/10.1109/ICPC58990.2023.00017
66. Vacchi, E., Cazzola, W.: Neverlang: a framework for feature-oriented language development. Comput. Lang. Syst. Struct. **43**, 1–40 (2015). https://doi.org/10.1016/j.cl.2015.02.001
67. Klint, P., van der Storm, T., Vinju, J.J.: On the impact of DSL tools on the maintainability of language implementations. In: Brabrand, C., Moreau, P.-E. (eds.) Proceedings of the of the Tenth Workshop on Language Descriptions, Tools and Applications, LDTA 2010, Paphos, Cyprus, March 28–29, 2010—satellite event of ETAPS, p. 10. ACM, New York (2010). ISBN 978-1-4503-0063-6. https://doi.org/10.1145/1868281.1868291
68. Åkesson, J., Ekman, T., Hedin, G.: Implementation of a Modelica compiler using JastAdd attribute grammars. Sci. Comput. Program. **75**(1–2), 21–38 (2010). https://doi.org/10.1016/j.scico.2009.07.003
69. Ekman, T., Hedin, G.: The JastAdd extensible Java compiler. In: Gabriel, R.P., Bacon, D.F., Lopes, C.V., Steele Jr., G.L., (eds.) Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada, pp. 1–18. ACM, New York (2007). ISBN 978-1-59593-786-5. https://doi.org/10.1145/1297027.1297029
70. Basten, B., van den Bos, J., Hills, M., Klint, P., Lankamp, A., Lisser, B., van der Ploeg, A., van der Storm, T., Vinju, J.J.: Modular language implementation in Rascal—experience report. Sci. Comput. Program. **114**, 7–19 (2015). https://doi.org/10.1016/j.scico.2015.11.003
71. Zarrin, B., Baumeister, H.: Design of a domain-specific language for material flow analysis using Microsoft DSL tools: An experience paper. In: Design of a domain-specific language for material flow analysis using Microsoft DSL Tools: An experience paper (2014)
72. van Beusekom, R., Groote, J.F., Hoogendijk, P.F., Howe, R., Wesselink, W., Wieringa, R., Willemse, T.A.C.: Formalising the Dezyne modelling language in mCRL2. In: Petrucci, L., Seceleanu, C., Cavalcanti, A. (eds.) Critical Systems: Formal Methods and Automated Verification—Joint 22nd International Workshop on Formal Methods for Industrial Critical Systems—and—17th International Workshop on Automated Verification of Critical Systems, FMICS-AVoCS 2017, Turin, Italy, September 18-20, 2017, Proceedings, volume 10471 of Lecture Notes in Computer Science, pp. 217–233. Springer, Berlin (2017). ISBN 978-3-319-67113-0. https://doi.org/10.1007/978-3-319-67113-0_14
73. Kurtev, I., Schuts, M., Hooman, J., Swagerman, D.-J.: Integrating interface modeling and analysis in an industrial setting. In: MODELSWARD, pp. 345–352. SciTePress, Setúbal (2017)
74. Raedts, I., Petkovic, M., Usenko, Y.S., van der Werf, J.M.E.M., Groote, J.F., Somers, L.J.: Transformation of BPMN models for behaviour analysis. In: Augusto, J.C., Barjis, J., Ultes-Nitsche, U. (eds.) Modelling, Simulation, Verification and Validation of Enterprise Information Systems, Proceedings of the 5th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems, MSVVEIS-2007, In conjunction with, pp. 126–137. INSTICC Press (2007). ISBN 978-972-8865-95-5

Olav Bunte is a teacher and PhD candidate in the field of computer science at the Eindhoven University of Technology, the Netherlands. He got his MSc degree at the same university and is close to completing his PhD.

Jasper Denkers is a PhD candidate in the field of programming languages at the Delft University of Technology, the Netherlands. He received his MSc degree in Computer Science from the Delft University of Technology in 2018. His research focuses on the industrial application of domain-specific languages developed with language workbenches.

Louis C. M. van Gool is Software Technology Developer at Canon Production Printing. He received his MSc and PhD degrees in Computer Science at Eindhoven University of Technology, in 2000 and 2006. After two years as a postdoctoral researcher, he concluded that his passion is to develop software technology with a strong theoretical base, but focussed on industrial applicability. After working for two years as a consultant for ICT NoviQ and two years as a product developer for Verum, he moved to Océ (currently known as Canon Production Printing), a company in his home town Venlo.

Jurgen J. Vinju is a senior researcher at Centrum Wiskunde & Informatica and a part-time full professor at TU Eindhoven. He is co-designer and maintainer of the Rascal metaprogramming language and co-owner of Swat.engineering BV.

Eelco Visser was Antoni van Leeuwenhoek Professor of Computer Science and chair of the Programming Languages Group at Delft University of Technology. His research was on the foundation and implementation of declarative specification of programming languages.

Tim A. C. Willemse is an associate professor and chair of the Formal System Analysis group at the Eindhoven University of Technology. He holds a part-time position as a senior research fellow at TNO-ESI. He received his MSc in 1998 and was granted a PhD in 2003, both from the Eindhoven University of Technology. His research interests cover the theory and semantics of process algebras and temporal logics, and algorithms for model checking and model-based testing. He has a proven track record of successful applications of Formal Methods to complex systems, which include the ERTMS Hybrid Level 3 standard and CERN's control software of the four main experiments at the Large Hadron Collider. He is the chair of the Industry Committee of Formal Methods Europe and is an active member of the Formal Methods research community.

Andy Zaidman is a full professor in software engineering at Delft University of Technology, The Netherlands. He received the MSc and PhD degrees in Computer Science from the University of Antwerp, Belgium, in 2002 and 2006, respectively. His main research interests include software evolution, program comprehension, mining software repositories, software quality, and software testing. He is an active member of the research community and involved in the organization of numerous conferences (WCRE'08, WCRE'09, VISSOFT'14 and MSR'18). In 2013 he was the laureate of a prestigious Vidi mid-career grant, while in 2019 he received the most prestigious Vici career grant from the Dutch science foundation NWO.