Extending Grammars and Metamodels for Reuse
The Reuseware Approach

Jakob Henriksson Florian Heidenreich Jendrik Johannes
Steffen Zschaler Uwe Aßmann

Technische Universität Dresden, Fakultät Informatik, D-01062
Dresden, Germany
{jakob.henriksson|florian.heidenreich|jendrik.johannes|
steffen.zschaler|uwe.assmann}@tu-dresden.de
February 28, 2008

Preprint of paper published in
IET Software Special Issue on Language Engineering, 2008.

Abstract

The trend toward domain-specific languages leads to an ever-growing plethora of highly specialized languages. Developers of such languages focus on their specific domains rather than on technical challenges of language design. Generic features of languages are rarely included in special-purpose languages. One very important feature is the ability to formulate partial programs in separate encapsulated entities, which can be composed into complete programs in a well-defined manner. This paper presents a language-independent approach for adding useful constructs for defining components. We discuss the underlying concepts and describe a composition environment and tool supporting these ideas – the Reuseware Composition Framework. To evaluate our approach we enrich the (Semantic) Web query language Xcerpt with an additional useful reuse concept—modules.

1 Introduction

The ever-increasing complexity of modern-day software development asks for the constant development of new languages to support specific tasks. Domain-specific languages (DSLs) are examples of such languages. For instance, the Semantic Web [1] is an area where many specialized declarative languages have been defined to precisely capture meaning and enable accurate description of data on the web. Examples of Semantic Web languages include ontology languages (e.g. OWL [2], RDF(S) [3], SWRL [4]) and query languages (e.g. XQuery [5], SPARQL [6], Xcerpt [7]). Software modeling is another important area where many declarative languages exists and where many DSLs have appeared, especially around the Unified Modeling Language (UML) [8].

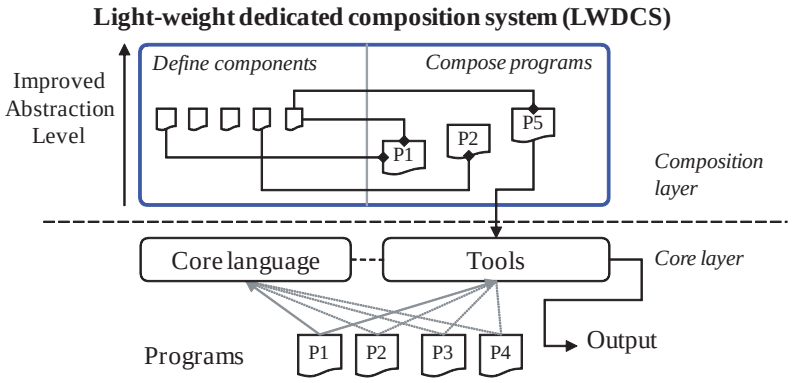

Figure 1: Layered composition system for additional abstractions.

These languages have been carefully designed and are very capable in their domains of operation. The language designers focus on the domain-specific concepts the languages are meant to support and develop them driven by specific requirements and use-cases. However, much remains to be done regarding the technical issues of language design. For example, some of these languages are lacking good concepts for abstraction and encapsulation; they have insufficient support for defining reusable *components*. On the other hand, it is well-known that there are many benefits to be harvested from component-based software development and this method is considered a vital part of large mature systems [9]. Yet, providing good support for abstractions and reuse concepts can be a quite complex task in its own right. Therefore, such support is often initially neglected when new languages are being developed and only added as an afterthought or at later stages in the development of the languages.

On the other hand, many fundamental abstraction concepts have a lot of properties that are independent of concrete languages. Hence, one can consider providing component-based development support on a language-independent level, without initially tampering with the specifics of each individual language. This has the advantage that the same composition techniques can be reused for many languages. More importantly, we claim that one can create a general tool framework for enabling component-based development for a number of languages lacking such capabilities. Using this general framework can make developing reuse abstractions for different languages more cost-efficient and convenient than having to augment each language individually.

Users of such augmented languages should, however, not have to deal with the technical intricacies of the reuse abstractions. Therefore, we aim for a *light-weight dedicated composition system* (LWDCS[1]) to be layered on-top of a particular *core* language and its associated tools that realize the required abstractions (see Figure 1). For example, in Figure 1, program `P1` is an instance of a particular (core) language and usable with the associated tools (e.g. editors, compilers, interpreters etc.). Using special operators from the LWDCS, this program can be decomposed into reusable components. This has the advantage that some of the components can be reused in

[1]Pronounced *low-deeze*.

different programs (e.g. program `P5`). The composed programs are instances of the core language and can be used with the core-language specific tools. The composition system is *dedicated* because it addresses issues for a single targeted language, and *light-weight* since once developed and deployed it is operable without its users directly being aware of it. The dedicated composition system allows for thinking of the new and richer abstraction constructs as first-class entities in the underlying language. Such a dedicated light-weight composition system can be developed using our composition framework—the Reuseware Composition Framework[2].

In our work we are mainly addressing declarative languages used in software modeling and on the Semantic Web. It should be noted that it is not straight-forward to transfer existing composition techniques and concepts to these declarative languages, since existing approaches are often based on black-box component models. For declarative languages, a pure black-box component model approach cannot be taken where components are used and composed according to well-defined inputs and outputs. One reason is that declarative languages are not only used to formulate encapsulated processing entities, like in many general-purpose languages, e.g. C++, C#, Java. Even though users of the language want to view declarative components as black boxes, the underlying composition technology cannot always do so. In order for declarative components to be reusable and deployable in different contexts, they need to be *opened up* such that they can properly be adapted into the new contexts. This is also true for any language where components are described on source-code level, e.g. as in aspect-oriented programming in Java. For a general-purpose language such as Java, one may view components at any abstraction level ranging from white-box to black-box. For most languages used in software modeling and on the Semantic Web, however, we do not have much choice.

In this work, the component model of a LWDCS, that is, the reuse abstraction that is deployed on top of a core language, is a gray-box component model based on program fragments. The foundations are provided by Invasive Software Composition (ISC) [10], a composition approach attractive for two reasons:

1. The provided composition technique is very general and is applicable to many different languages and situations.

2. It is flexible wrt. the granularity of components and symmetry of composition [11] and can, thus, capture and realize many well-known composition approaches (e.g. aspect orientation [12] and hyper space composition [13]).

Thus, the generality of ISC as described above gives us an appropriate foundation to build upon and to specialize according to different situations and needs.

In this paper we describe and discuss the following issues:

- We give a formal explanation of how the gray-box, fragment-based, composition technique provided by ISC can be made available for arbitrary languages.

- We present a framework and tool that can make the general composition technique available for an arbitrary language given its grammar or metamodel.

[2] Available at http://www.reuseware.org.

- We describe how the framework allows for specific requirements of certain languages to be addressed by developing special-purpose composition operators dealing with these issues.

This paper is an extension of our previous work [14, 15]. Here we extend our work by making the full connection between language extensions (to enable the composition technique from ISC) and dedicated composition systems (LWDCS). This includes explaining how specialized composition operators may be developed for specific languages and how component models should be refined to accommodate them.

1.1 Problem Scenarios

In the following we give examples of important issues of DSLs in the fields of the Semantic Web and in software modeling that need to be addressed in the near future. These are also issues we address using the techniques and tools presented in this paper.

The Semantic Web

At the core of the Semantic Web lies a notion of duality: data annotation and its access. Annotating data with semantical information results in the possibility for software agents to process and to make better sense of the otherwise "dead" data. Accessing the annotations allow for the agents to draw conclusions and reason over the underlying data in order to make decisions, for example when realizing complicated web-services. To annotate data, metadata languages are needed (ontology-based languages), and for accessing the data query languages must be available. As the use of semantic information on the web becomes common-place, ontologies will inevitably grow in size and queries will become more complicated due to the existence of, and the need to integrate and process, many different data schemata, in contrast to the single-schema situation in traditional databases.[3] At the same time, an incredible amount of different languages are being developed for these different purposes, but few actually bother to provide sufficient levels of abstraction and reuse constructs as first-class concepts. To tackle this situation, a more generic approach of realizing component-based development must be made available on the Semantic Web.

Software modeling

As Model-driven Software Development (MDSD) are playing an increasingly important role in developing software systems it is crucial to provide means to compose models and transform them, both from more abstract to less abstract representations (vertically) and to different representations on the same level of abstraction (horizontally). It is obvious that new languages and tools for transformation, composition and weaving of models are needed. Transformation languages like Query / View / Transformations (QVT) [16] and ATL [17] are addressing these needs. Unfortunately, these languages do not provide built-in concepts for reuse, something which is vital to tackle complexity. This holds especially true in product-line engineering where models are

[3] The Web is inherently heterogeneous.

utilized for expressing core and variable parts of software systems. The models are growing in size and are often formulated upon different metamodels. While the first issue demands for increased scalability, the latter results in more complex model transformations. One feature of metamodels and transformation languages that alleviates complexity in models and model transformations is built-in support for reuse. Therefore it is useful to introduce a generic approach for realizing reuse in a—preferably—language-independent way.

The remainder of this paper is structured as follows. Section 2 deals with preliminaries, motivating why we build upon ISC and introduces Xcerpt in more detail. In Section 3 we explain how we can enable the ISC composition technique for any formal (context-free) language through grammar extension. In Section 4 we make the connection between such extended language grammars and component models in dedicated composition systems realizing abstractions for languages. As an example of such an abstraction we introduce modules into Xcerpt. To properly realize such a dedicated composition system we explain what dedicated composition operators are in Section 5 and describe how they can be defined. Section 6 presents the implemented framework which can be used to develop the dedicated composition systems. Finally, in Section 7 related work is referenced and Section 8 concludes the paper and presents directions for future work.

2 Preliminaries

Before we describe our formalism in Section 3, we give a brief introduction to ISC and motivate why we build upon, and extend, ISC. As we are going to apply and demonstrate our techniques on the Web query language Xcerpt, we introduce the language in the second part of this section.

2.1 Invasive Software Composition

ISC is a grey-box composition approach where components—or fragment components[4]—are static, source-code entities with well-defined interfaces using the notion of *hooks*. A hook is essentially a location in a component which may be replaced by another component by using ISC composition operators. As such, the hooks of a component define its composition interface. Intuitively, the replacement of a hook with some existing component constitutes the basic composition technique of ISC. This rather loose and general composition technique has both advantages and disadvantages. The advantage is that the composition technique is very general and realizable for any language used to author the components. In particular, in Section 3 we describe how an arbitrary language can be made subject to this composition technique. The disadvantage is that the technique is primitive in its description, that is, application of the technique only constitutes a *low-level composition step*. The realization of a needed abstraction will often require a set of such low-level composition steps to be executed

[4]We will make no difference between the terms fragments and components in this paper—a component is a fragment component (or simply fragment).

as a unit in a *high-level composition step*. That is, describing compositions on the level of ISC is cumbersome and provides no level of reuse of commonly used operations. We will return to this issue in Section 4 where we discuss composition operators. In particular, we will address and explain how we can remedy this short-coming in Section 5. As such, we exploit the versatility of the available composition technique, but improve upon its limitations.

2.2 Xcerpt

Xcerpt is an XML and RDF query and transformation language. In contrast to similar languages like XQuery [5] and XSLT [18], Xcerpt follows the logic programming paradigm (rule-based and declarative) and clearly separates query and construct parts of programs.

An Xcerpt program consists of a finite set of Xcerpt *rules*. The rules of a program are used to derive new (or transform) XML data from existing data (i.e. the data being queried). In Xcerpt, two different kinds of rules are distinguished: *construct rules* and *goal rules*. Construct rules are used to produce intermediate results and takes the form: `CONSTRUCT head FROM body END`. Goal rules make up the output of programs and looks like: `GOAL head FROM body END`. Intuitively, the rules are to be read: if `body` holds, then `head` holds. Formally, `head` is a *construct term* and `body` is a set of *query terms* joined by some logical connective (e.g. `or` or `and`). A rule with an empty body is interpreted as a fact, i.e. the rule head always holds.

While Xcerpt works directly on XML data, it has its own data format for modeling XML documents. Xcerpt *data terms* model XML data. Data terms use a square bracket notation but there is a one-to-one correspondence between the two notions. The data term `book [ title [ "White Mughals" ] ]` and the XML snippet `<book><title>White Mughals</title></book>`, for example, model the same data. The data term syntax makes it easy to reference XML document structures in queries.

Xcerpt *query terms* are used for querying data terms and intuitively describe patterns of data terms. Query terms are used with a pattern matching technique to match data terms.[5] Query terms can be configured to take partiality and/or ordering of the underlying data terms into account during matching. Square brackets are used in query terms when order is of importance, otherwise curly brackets may be used. E.g. the query term `a [ b [], c [] ]` matches the data term `a [ b [], c [] ]` while the query term `a [ c [], b [] ]` does not. However, the query term `a { c [], b [] }` matches `a [ b [], c [] ]`, since ordering is said to be of no importance in the query term. Partiality of a query term can be expressed by using double instead of single brackets. Query terms may also contain logic variables. If so, successful matching with data terms results in variable bindings used by Xcerpt rules for deriving new data terms. For example matching the query term `book [ title [ var X ] ]` with the XML snippet above results in the variable binding `{X / "White Mughals"}`. *Construct terms* are essentially data terms with variables. The variable bindings produced by query terms in the body of a rule

[5] *Simulation unification*, see [19] for details of this technique.

can be applied to the construct term in the head of the rule in order to derive new data terms. In the rule head, construct terms including a variable can be prefixed with the keyword `all` to group the possible variable bindings around the specific variable.

```
GOAL                                                                    1
  authors [ var X ]
FROM                                                                    3
  book [[ author [ var X ] ]]
END                                                                     5

CONSTRUCT                                                               7
  book [ title [ "White Mughals" ], author [ "William Dalrymple" ] ]
END                                                                     9
```

Listing 1: The construct rule defines some data about books and their authors and the goal rule queries this data for authors.

An example Xcerpt program querying a bibliography fact base is shown in Listing 1, resulting in the data term `authors [ "William Dalrymple" ]`. For a more complete and in-depth introduction to Xcerpt, please consult [7].

The reuse abstraction directly available in Xcerpt is on the level of rules. Rules can be chained together where the output from one rule is used as input to another. As such, rules may be reused and configured together in new ways. Other kinds of reuse are not offered. For example, there is no way to reuse query terms or any other smaller syntactical entities supported by the language. Neither is it possible to reuse larger sets of rules, each of which contribute to supply some required service. We will remedy this situation by introducing a new abstraction in Xcerpt in Section 4.1: *modules*, which will allow Xcerpt programmers to define reusable rule sets. The module concept will be offered by a LWDCS, based on the fragment component model of ISC.

3 Formalism – Grammar Extensions for Composition

In this section we describe how it is possible to enable the versatile composition technique of ISC for any formal language. This means that we define how a language can be extended as so to be able to describe fragments to allow for the technique to operate on them.

Before defining our notion of fragments and how they can be specified in more detail, we briefly look at context-free grammars, as the principal formalism for describing the syntax of formal languages. Fragment components and ISC can also be used for languages based on metamodeling, but initially we focus on grammar-based languages to simplify our explanations.

Formally, a context-free grammar (CFG) is a 4-tuple [20]:

$$G = (V_t, V_n, P_r, S)$$

where V_t is a finite set of terminals, V_n a finite set of non-terminals, P_r a finite set of production rules $V_n \rightarrow (V_t \cup V_n)^*$ and $S \in V_n$ the start symbol. Each production rule $V_n \rightarrow (V_t \cup V_n)^*$ can be used to rewrite V_n by $(V_t \cup V_n)^*$. A language L is context-free if there exists a context-free grammar G that generates it. Intuitively, a

context-free grammar G of a programming language L defines a (possibly infinite) set of programs that conform to G. Most programming languages can be defined by a context-free grammar, and we only deal with such languages here.

```
XcerptProgram      = XcerptStatement+;                                    1
XcerptStatement    = GoalQueryRule | ConstructQueryRule;
                                                                          3
ConstructQueryRule = "CONSTRUCT", ConstructTerm, ("FROM", QueryTerm)?,
                       "END";                                             5
GoalQueryRule      = "GOAL",      ConstructTerm, ("FROM", QueryTerm)?,
                       "END";                                             7

QueryTerm          = StructuredQt | ...                                   9
ConstructTerm      = ...
```

Listing 2: A selection of the production rules for Xcerpt.

EBNF [21] is a textual notation for context-free grammars. As an example, the production rules P_r of a context-free grammar describing the syntax of a subset of the Xcerpt language is given in Listing 2 in EBNF syntax (based on a grammar given in [22]). The symbols `?` and `|` have their standard EBNF meaning. The start symbol for this grammar is assumed to be `XcerptProgram`. Intuitively, this means that any valid program of the grammar can be derived starting from the symbol `XcerptProgram` by successively applying production rules until no more non-terminals are contained in the resulting string.

A *program* P of a language L defined by the grammar G is a set of syntactically well-formed statements wrt. G. More specifically, the program P can be derived from G using its defined production rules starting from the start symbol S. We introduce the notion of *grammatical types*, which forms the foundational formalism for our notion of *type-safe* compositions. We therefore say that P is of grammatical type S.[6]

Definition 1 (Grammatical types) *Given a string T and a context-free grammar G, every non-terminal $v_n \in V_n$ that T can be derived from is a* grammatical type *(wrt. G) of T.*

As a consequence, the grammatical type of any program derived from G is S, the start symbol of G. For example, for the grammar in Listing 2, we say that the type of a valid program is `XcerptProgram`. According to Definition 1, strings can have more than one grammatical type. In particular this is the case when a grammar G contains production rules with choices, for example, on Line 2 in Listing 2 where the non-terminal `XcerptStatement` is defined. A string that can be derived from the non-terminal `GoalQueryRule` can also be derived from the non-terminal `XcerptStatement`.

As a further example, consider the program in Listing 3. The program queries a bibliography database `biblio.xml`, extracts information about titles and authors and then constructs the result in a specific way, as seen in the construct term of the goal rule. The string representing the program can be generated starting from three different non-terminals in the grammar in Listing 2: `XcerptProgram`, `XcerptStatement`, `GoalQueryRule`. Hence, its set of grammatical types includes these non-terminals.

[6] Whenever we refer to the *type* of a fragment, we shall mean its grammatical type.

```
GOAL
    results [ all result [
        var Title, all var Author
    ] ]
FROM
    in { resource { "file:biblio.xml", "xml" },
      bib [[  book [
          var Title -> title [[ ]],
          authors [[
            var Author -> author [[ ]]
          ]]
      ] ]]
    }
END
```

Listing 3: An Xcerpt program querying a bibliography database for authors and titles.

In ISC, programs are composed from fragments of code, or so-called *partial programs*. Partial programs are not complete programs in themselves, but can describe specific concerns in more complete programs. Partiality of a program can stem from two different sources.

```
in { resource { "file:biblio.xml", "xml" },
  bib [[ book [
      var Title -> title [[ ]],
      authors [[
        var Author -> author [[ ]]
      ]]
  ] ]]
}
```

Listing 4: A partial Xcerpt program: a query term.

First, given a grammar G one may specify partiality of a program P wrt. G by exchanging the start symbol S of G to $S' \in V_n \setminus \{S\}$. By using a start symbol other than S, we effectively derive a new grammar G', defining a sub-part of a valid G-program. Such a partial program is, consequently, of grammatical type S'. For example, Listing 4 is a partial program wrt. the original Xcerpt grammar. In fact it is the Xcerpt query term from Listing 3 and the type `QueryTerm` belongs to its set of grammatical types. Therefore, it is a valid program wrt. the original Xcerpt grammar where the start symbol has been changed to `QueryTerm`.

Second, partiality of a program can also come from within a specific sub-part as it was defined above. A partial program may also be underspecified "inside"; that is, at a deeper nesting level. For example, a partial program consisting of a goal rule in Xcerpt might be underspecified by leaving out the query term, thus allowing the rule to be configurable wrt. the query term (see Listing 5). To allow for such underspecifications, we introduce the notion of a *variation point*. A variation point is a place-holder for some partial program that is still unspecified.

Definition 2 (Variation point) *A* variation point $v(v_n, I)$ *represents the uninstantiated non-terminal* $v_n \in V_n$ *from a grammar* G*. The grammatical type of a variation point* $v(v_n, I)$ *is* v_n*.* I *is an identifier associated with the variation point.*

Listing 5 shows the goal rule from Listing 3, where we have replaced the concrete

query term with a variation point for the non-terminal `QueryTerm`: `<<myVarPoint : QueryTerm>>`. This allows us to vary the query term used in this rule, or seen another way, allows us to reuse the desired query term in other rules. Here we use the character sequences << and >> to markup the variation point and `myVarPoint` is the identifier for the specific variation point.

```
GOAL
    results [ all result [                                    2
        var Title, all var Author
    ] ]                                                       4
FROM
    <<myVarPoint : QueryTerm>>                                6
END
```

Listing 5: An underspecified partial Xcerpt program with one variation point.

To enable processing of partial programs containing variation points, we need to extend their grammar to include syntax for the variation points. Therefore, we introduce variation point syntax as non-terminals (we use V_v to denote a set of variation points):

Definition 3 (Context-free reuse grammar) *A* context-free reuse grammar *for a context-free grammar $G = (V_t, V_n, P_r, S)$ is a context-free grammar*

$$G_I = (V_t, V_n \cup V_v, P_{rI}, S_I)$$

transformed via the function $G_I = \tau(G)$ where $S_I \in V_n$ (possibly $S_I \neq S$) and for each $vp \in V_v$ there is a non-terminal $n \in V_n$ such that $vp = v(n, I)$, v is a variation point for n. For any reuse grammar G_I, we call G the core grammar *of G_I.*

τ fulfils two properties. First, τ is preservative*, meaning that any string that can be derived from S_I wrt. G can still be derived from S_I wrt. grammar G_I. Second, τ is* type preservative*. This means that τ transforms the production rules P_r of G such that each $vp \in V_v$ is introduced in rules of P_{rI} with the requirement that vp is only an alternative for its corresponding $n \in V_n$ (according to $vp = v(n, I)$).*

The transformation function τ, thus, extends a core grammar for a language $\mathcal{L}$ into a corresponding reuse grammar describing a language used for writing partial programs (fragments) of $\mathcal{L}$. The composition technique of ISC can thus be applied to such fragments. τ is a generalizing grammar transformation in the sense of [23]. Intuitively, if variation points for some grammatical type are introduced, we only extend the production rules of the core grammar such that the variation points become valid alternatives for partial programs of that type.

Thus, a partial program may be specified by freely choosing a start symbol $S_I \in V_n$ and using a set V_v of variation points for a subset of V_n.

Listing 6 is provided as an example of how the production rules of the (core) Xcerpt grammar from Listing 2 are transformed via τ to allow for replacing a concrete query term with a variation point. In the production rules defining `ConstructQueryRule` and `GoalQueryRule`, the reference to `QueryTerm` is replaced by a grammatical choice. Such a choice allows for the specification of a variation point—*v(*`QueryTerm`*,I)*—as an alternative to a concrete query term.

```
XcerptProgram      = XcerptStatement+;                                          1
XcerptStatement    = GoalQueryRule | ConstructQueryRule;
                                                                                3
ConstructQueryRule = "CONSTRUCT", ConstructTerm,
                       ("FROM", ( QueryTerm | v(QueryTerm,I) ))?, "END";       5
GoalQueryRule      = "GOAL",      ConstructTerm,
                       ("FROM", ( QueryTerm | v(QueryTerm,I) ))?, "END";       7

v(QueryTerm,I)     = "<<", I, ":", "QueryTerm", ">>";                           9

QueryTerm          = StructuredQt | ...                                         11
ConstructTerm      = ...
```

Listing 6: Reuse grammar production rules including a variation point for `QueryTerm`.

In general there are two basic approaches for deriving a context-free reuse grammar G_I from a context-free core grammar G via τ:

1. Introduce variation points in V_v of G_I for every non-terminal in V_n of G (i.e. $|V_n| = |V_v|$ and $\forall v_n \in V_n : \exists v(v_n, I) \in V_v$). In this case, we call G_I a *universal extension* of G.

2. Introduce variation points for a well-chosen subset of non-terminals in V_n of G (i.e. $V_v \subset V_n$). In this case, we call G_I a *tailored extension* of G.

We argue that tailored extensions are more interesting since they can provide composition opportunities for specialized purposes. Furthermore, they disallow the variability of certain constructs in fragments, thus supporting encapsulation. We will make use of such a tailored extension when we develop a dedicated composition system for Xcerpt. We will return to these issues in Section 4.

Definition 4 (Fragment) *A* fragment *is a partial program wrt. a context-free grammar G and a valid program wrt. a context-free reuse grammar $G_I = \tau(G)$. The grammatical type of such a fragment is S_I.*

The variation points contained in a fragment form its *composition interface*. Variation points in such a composition interface can be subject to composition, that is, they can be replaced by other fragments. The composition technique used during composition is subject to *type restrictions*. That is, it is enforced that the type of a variation point and the type of the fragment to replace it, match. Definition 5 specifies what type restrictions are enforced and what matching of grammatical types mean. It should be clear that the types involved are derived from the underlying core grammar and its set of non-terminals.

Definition 5 (Type safety) *Let G be a context-free core grammar, F_1 and F_2 fragments valid wrt. $G_I = \tau(G)$ and $v(v_n, I)$ a variation point in F_1. Let $GT_2 \subseteq V_n$ be the set of grammatical types of F_2. Then the composition technique can be applied to F_1, F_2, and $v(v_n, I)$, iff $v_n \in GT_2$; that is, if the grammatical types of the variation point and the replacement fragment match.*

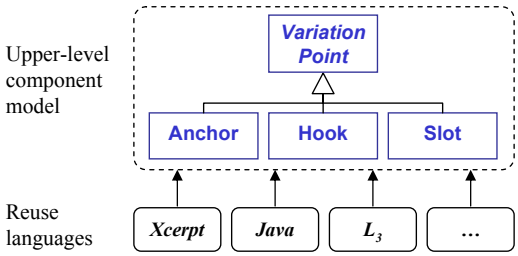

Figure 2: A common upper-level component model, reused across languages and allowing for language-independent composition executions.

For example, the grammatical types of the query term in Listing 4 are {`QueryTerm`, `StructuredQt`}. Replacing the variation point in Listing 5 with the fragment in Listing 4 is type safe. This is the case since the type of the variation point belongs to the set of grammatical types of the fragment (`QueryTerm` $\in$ {`QueryTerm`, `StructuredQt`}). Trying to replace the same variation point with the fragment in Listing 3 will result in a type error (since `QueryTerm` $\notin$ {`XcerptProgram`,`XcerptStatement`,`GoalQueryRule`}). By enforcing type safe composition, it is guaranteed that the result of a composition will always be a valid partial program wrt. the underlying core grammar.

3.1 Role of Metamodels for composition

EBNF is just one notation for describing (context-free) formal languages. Another formalism for describing languages is provided by metamodeling. Languages such as EMOF [24] and Ecore [25] can be employed for this purpose.[7] From a language specification perspective, describing languages in EBNF and its textual syntax is arguably easier, but metamodeling provides higher expressiveness and more flexibility. For example, modern tools and software development frameworks such as the Eclipse Platform[8] provide good support for metamodeling, model-based code-generation for parsers etc. Metamodeling also provides the possibility to reference *upper-level metamodels* where common concepts can be reused across metamodels (see Figure 2).

Figure 2 shows upper-level component model modeling concepts reused in every dedicated composition system. An example of such a concept is the notion of a *variation point*. There are also other refined types of variation points, e.g. *slots*, *hooks*, and *anchors*. A *slot* is a variation point that can be replaced by a fragment once. A *hook* is a variation point that can be replaced by a fragment multiple times. An *anchor* is another kind of variation point for accessing fragments within fragments, a notion important in model composition (see Section 4.2). The advantage with using upper-level models is that tools can be implemented where compositions are executed by only inspecting fragments' references to the upper-level model. As such, the tooling can partly be realized language independently, a great advantage from an implementation perspective.

[7] When we refer to a *metamodel*, we shall mean an EMOF/Ecore metamodel.
[8] Via the Eclipse Modeling Framework [25].

3.1.1 Grammars as Metamodels

To move from grammars to metamodels we need a generation methodology for this transition. For language descriptions, we make a clear separation between the abstract and concrete syntax. This is motivated by the fact that several concrete syntaxes can be mapped to one abstract syntax, which is the case for Xcerpt, for example. Furthermore, such a separation is convenient since the abstract syntax specification of a language directly corresponds to its hierarchy of grammatical types. Here, we are going to focus on abstract syntax.

We choose a two-phased generation approach (a similar approach is presented in [26]) consisting of a *normalization phase* and a *generation phase*.

1. **Normalization phase** This phase re-writes the grammar in preparation for being transformed into a metamodel.

   (a) Production rules that contain combination of choices and aggregations are split into several rules, such that each rule either contains only choices (`|`) or aggregations (`,`). This process is also referred to as *rule splitting*. The new (non-terminal) constructs are named `Aggregation` (resp. `Choice`) followed by the original construct name and a unique number.

2. **Generation phase** This phase transforms each kind of production rule in the normalized grammar into a part of the resulting metamodel.

   (a) *All rules.* Each production rule head (that is, left-hand-side non-terminal) is translated into a metaclass named after the name of the rule head.

   (b) *Choice rules.* Grammar production rules containing only choices are translated into inheritance relationships with the rule head metaclass as parent class.

   (c) *Aggregation rules.* Grammar production rules representing aggregations are translated into composite aggregations between the right-hand-side metaclasses and the rule head metaclass as the composite metaclass.[9]

   (d) *Terminal rules.* If the (non-terminal) symbol corresponds to a set of character strings, integer values, or boolean values, then an attribute is added to the relevant metaclass with the corresponding primitive type (i.e. `String`, `Integer`, or `Boolean`). Otherwise, the symbol is translated into featureless metaclass named after the symbol.

All metamodels that result from the above transformation have the following properties: 1) Inheritance is only utilized to express grammatical types, not for feature inheritance. No parent metaclasses have features (i.e. attributes or operations). 2) All aggregations in the generated metamodel are composite aggregations. Hence, every instance of the metamodel has a tree structure. Figure 3 exemplifies this approach by showing a metamodel corresponding to the abstract syntax part of the Xcerpt reuse grammar from Listing 6.

[9] Since we focus on abstract syntax, sequentiality of aggregations in grammars does not need to be preserved.

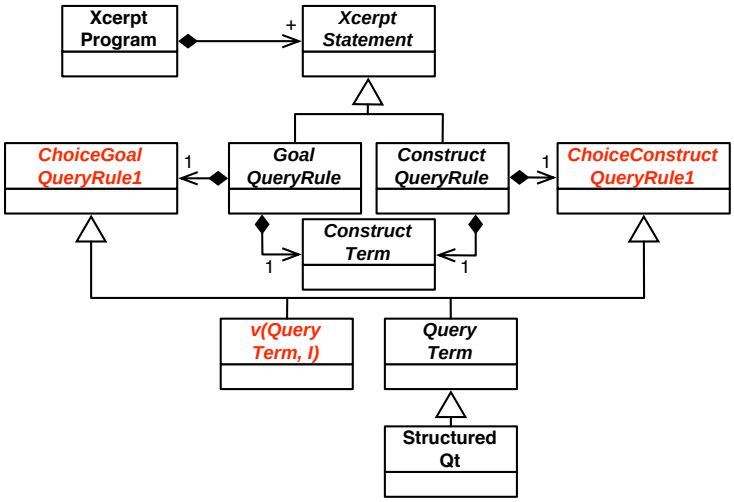

Figure 3: Reuse metamodel corresponding to the (abstract syntax) reuse grammar in Listing 6 after being transformed according the two-phased grammar-to-metamodel algorithm.

4 Component Models and Beyond

In Section 3 we introduced the formalism for enabling ISC to be applied to any language. That is, we explained how it is possible to extend a language grammar (or metamodel) such that certain language constructs are *variable* and their instances can be left unspecified in reusable components. The composition technique of ISC can thus be applied to such components.

Composition systems Before introducing composition operators that exploit the composition technique that can be made available for arbitrary languages, we re-connect with the idea of composition systems as introduced in Section 1 and visualized in Figure 1. Recall that we realize additional abstractions for different languages via composition.

A composition system can be seen as a triple consisting of a *composition language*, *component model* and *composition technique* [10]. We shall deal with composition languages a little later, but as already mentioned, the composition technique we build on is the one from ISC. The component model must properly describe what components look like and how they may be accessed. It should be clear that, on the level of ISC, the required component model for a particular language is captured in the reuse grammar as formally introduced in Section 3.

We argue that a dedicated composition system should be constructed as a refinement of a generic composition system, where the triple comprising the system is specialized for the task at hand - indeed tailored (see Figure 4). As can be seen from Figure 4, the dedicated composition language is adapted for the specialized task and refined from a more general-purpose language. Furthermore, the dedicated component

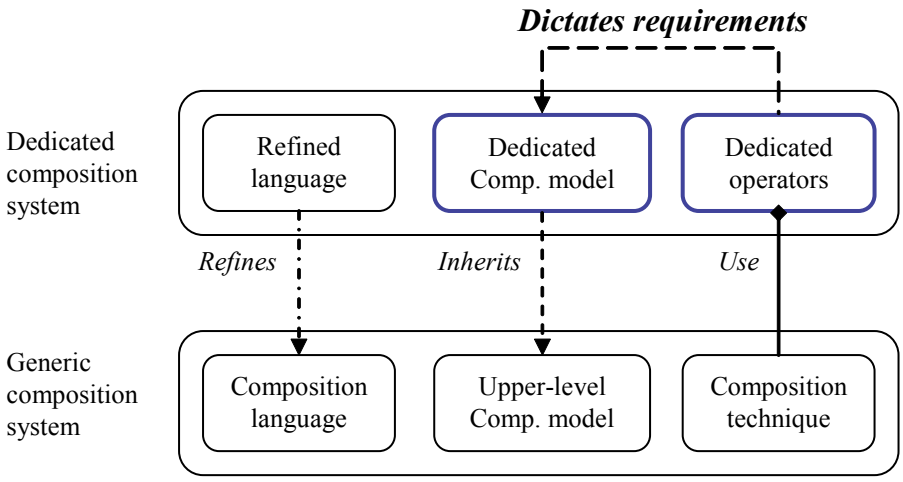

Figure 4: A dedicated composition system realizing a specific abstraction construct is created as a refinement from a generic composition system.

model references an upper-level component model where general (invasive) composition system concepts are modeled. Finally, instead of including a general composition technique and generic operators in the dedicated composition system, it is shipped with a set of predefined, specialized, composition operators. We shall now spend some time discussing the available composition operators in ISC.

Composition operators In order to use components and to put them together into useful programs (or models), we need composition operators implementing the composition technique. ISC distinguishes two *primitive composition operators* that differ in how they use the provided composition technique.

1. *Bind* replaces a slot in a fragment with a suitable fragment once, disabling the possibility of further applying operators on the slot. That is, the slot is removed.

2. *Extend* replaces a hook in a fragment with a suitable fragment once, but enables the possibility to apply other operators on the same hook. That is, the hook remains in the fragment.

Both primitive operators realize the type-safe composition technique (see Definition 5).

An implementation of the primitive operators transforms abstract syntax trees (ASTs) of fragments. In these trees each variation point is a node, and each fragment is a tree with a clearly identified root node. If the primitive operators are applied on ASTs, that is, on fragments written in a language that is described by a context-free grammar, then the realization of the operators can be described as follows:

1. *Bind* replaces a node in an AST that represents a slot with the root node of a suitable fragment. Because fragments are always trees, the fragment tree can

not be bound to several variation points. Thus, if the same tree is bound several times, then it is copied.

2. *Extend* inserts the root node of a suitable fragment as a sibling to the hook node in a tree. As such, the operator enables the possibility of applying other operators on the same hook. That is, the hook remains in the fragment. Again, if the same tree is bound several times, it is copied.

The primitive composition operators can be used in two ways. First, they can be used in *composition programs* to describe compositions of fragments. To formulate such composition programs we need a *composition language*. Since the primitive operators exist independently of concrete core languages, such a composition language can be generic. Furthermore, it can assume different forms—for example, it can be script-like or declarative, either with textual or graphical syntax. Second, they can be used to realize specific abstraction concepts for certain languages. This is achieved by using the primitive operators for implementing suitable composition operators in the dedicated composition system addressing these issues.

Even realizing simple abstractions can require components to be modified and transformed in several ways and in different locations. The intention of a dedicated composition system is to allow users to think of the additional abstraction constructs as first-class entities. Thus, if its realization requires a collective effort by several primitive composition operators, it is of importance to be able to encapsulate that sequence of operators as an atomic reusable unit. We will call such an atomic unit a *complex composition operator*.

The introduction of complex composition operators requires our notion of component model to be defined accordingly (see relationship between *dedicated operators* and *dedicated component model* in Figure 4). In particular we need to combine the notions of a composition program and an expression in the reuse language. In order to give users the impression of a new first-class construct, complex composition operators must be invokable directly from a program in the reuse language. Additionally users must not be required to indicate target variation points as explicit parameters to these calls. We therefore introduce the notion of *inline composition*. This means that complex composition operators invoked from a reuse language program implicitly define a variation point, which they also modify.

Before we look at these issues in detail, we walk through two examples. First, we present an application of our techniques to Xcerpt. We will introduce the notion of a *module* in Xcerpt and explain how this abstraction—the possibility of reusing useful rule-sets across programs—can be realized using our composition technique. Second, we show an example of aspect weaving for models.

4.1 Modules for a Web Query Language – Xcerpt

Xcerpt does not, at the time of writing, allow for a set of rules to be collected into a reusable module, like in other logic programming systems, e.g. XSB[10]. Here we

[10] http://xsb.sourceforge.net

describe how this can be achieved by realizing the module concept as a first-class construct in a LWDCS layered on top of Xcerpt and its query engine (see Figure 1). We refer to [27] for more details on Xcerpt modules.

```
MODULE rdfsEngine_subClassOf
CONSTRUCT
  public rdfsengine [
   output [
    inferredSubClassOf [
       all subClassOf [ var Subclass, var Superclass ]
    ]  ]  ]
FROM
  or {
     declsubclassof [ var Subclass, var Superclass ],
     and {
       declsubclassof [ var Subclass, var Z ],
       declsubclassof [ var Z, var Superclass ]
     }  }
END

CONSTRUCT
  declsubclassof [ var Subclass, var Superclass ]
FROM
  public rdfsengine [
   input [
     explicitsubclassof [ var Subclass, var Superclass ]
  ]  ]
END
```

Listing 7: An Xcerpt module in the file `/subclassof.mxcerpt` for inferring implicit subclass-of relationships in an ontology.

Modules allow developers to encapsulate a set of rules as a unit such that they may be reused across programs and users. Without having modules as first-class constructs in the language and having tool support to handle them, the only possibility of reusing rules is by copy-and-paste between programs. This complicates maintenance and sharing of rules on the Web.

```
IMPORT /subclassof.mxcerpt AS rdfsengine

GOAL
  hassubclass [ all var Super ]
FROM
  in rdfsengine (
   rdfsengine [[
    output [[
     inferredSubClassOf [[
       subClassOf [ var Sub, var Super ]
     ]]  ]]  ]]
  )
END

CONSTRUCT
  to rdfsengine (
   rdfsengine [
    input [
     explicitsubclassof [ var Subclass, var Superclass ]
    ] ]
  )
FROM
  owl [[
   Class {
```

```
      id { var Subclass },
      subClassOf {                                                    26
       attributes { about { var Superclass } }
      }                                                               28
    }  ]]
END                                                                   30

CONSTRUCT                                                             32
   owl [
     Class [ id [ "SportsEquipment" ] ],                              34
     Class [ id [ "TennisRacket" ],
             subClassOf [ about [ "SportsEquipment" ] ] ],            36
     Class [ id [ "WilsonTennisRacket" ],
             subClassOf [ about [ "TennisRacket" ] ] ]                38
   ]
END                                                                   40
```

Listing 8: An Xcerpt program making use of the module

As an example, we show an Xcerpt module providing simple reasoning capabilities for ontology documents. Ontologies are nowadays commonly used on the Semantic Web for modeling domain information. A common use of such ontologies is to arrange the central concepts of the modeled domain in a class hierarchy. Ontology reasoners are often employed to infer information contained in such ontologies, e.g. to derive implicit subclass-of relationships.

The rules in Listing 7 describe a reusable Xcerpt module, which can be used as a simple inference engine for computing such implicit information without employing the full force of an ontology reasoner. This Xcerpt module is stored in the file `/subclassof.mxcerpt`.

The second rule in Listing 7 functions as an *input rule* to the module, creating a view over the data which is to be reasoned on. The first rule, on the other hand, computes the implicit information contained in the view of the second rule, and as such functions as an *output rule* for the module. The *input* and *output* nature of the two rules is reflected by the introduced `public` keyword in the query and construct terms of the rules, respectively. When not specified, the visibility of the terms is drawn from the default value on Line 1 (here `private`). The visibility of rule terms are used to enforce *module encapsulation*, an important property we will return to shortly.

The program in Listing 8 makes use of the module shown above and consists of three rules. The first (goal) rule makes up the output of the program, the second rule constructs a view over an ontology to be provided to the module and the third rule encodes a (simplified) OWL [2] ontology as a fact (the rule lacks a body). For the sake of simplicity, the ontological data is directly encoded in the program. The given ontology describes sports articles and their subclass-of relationships.

Several constructs that do not belong to Xcerpt, but are injected into the reuse language, are used in the module and the program above. In the module, these are the module declaration construct (Line 1) and the visibility constructs (`private` and `public`). In the importing program, these are the module import construct (Line 1), the *in-module* construct (Line 6) and the *to-module* construct (Line 16). The module import construct declares a module, associates it with a short convenient name and makes the rules from the module available to the importing program (`IMPORT`). The *in-module* construct queries data from the module and the *to-module* construct produces

input data for the module.

As a module consists of a set of rules, they should all be included in the importing program at composition-time, such that they are available to the Xcerpt interpreter when the composition result is executed. However, properly realizing the module system is more subtle and complicated than just executing the merger of different rulesets. In order to allow modules to be encapsulated, one must ensure that inappropriate rule dependencies do not occur when programs and modules are merged and executed. That is, programs should only have access to certain rules in imported modules, and vice versa. This encapsulation can be realized by transforming the heads and bodies of the rules of the imported module in appropriate ways.

In fact, the introduced constructs (*import*, *in-module* and *to-module*) are specifically developed complex composition operators and the visibility constructs (`public` and `private`) are used as mark-up to guide the operators at composition-time such that the encapsulation of the modules is properly realized.

Listing 9 shows two rules that can be used to test the encapsulation properties of the module system when added to the rules in Listing 8. Clearly the second rule in the module (Listing 7) constructs data terms with the label `declsubclassof`. The first goal rule above tries to access the internal module data by simply matching the label. The second rule attempts the same, but uses the provided *in-module* composition construct. When executing the rules above using the Xcerpt engine, the result from the first rule is `No results`, while the second rule produces the data term `access_allowed []`, as expected.

The details of how the module system is implemented are discussed in Section 5.

```
GOAL
 intrusion_achieved []
FROM
 declsubclassof [[ ]]
END
```

```
GOAL
 access_allowed []                    2
FROM
 in rdfsengine (                      4
    declsubclassof [[ ]]
 )                                    6
END
```

Listing 9: Testing the encapsulation capabilities of the new module system.

4.2 Aspect Weaving for a Modeling Language – Ecore

Instead of first extending a grammar and then transforming it to a metamodel as shown in the previous section, we can also consider directly applying the language extension formalism on metamodels. This can be done for metamodels that conform to the structural implications of the metamodels transformed from grammars (see Section 3.1).

In this section we illustrate the usefulness of metamodel extension to realize separation of concerns in form of aspects in modeling. Aspect-Orientated Programming (AOP) [12] is a well-known reuse formalism supporting separation of cross-cutting concerns in software systems. Code realizing these concerns can be kept separate from the core implementation and instead woven into the (object-oriented) implementation upon request. Examples of systems supporting such techniques are AspectJ [28] and AspectC++ [29]. Here we show how a similar technique can be realized for the mod-

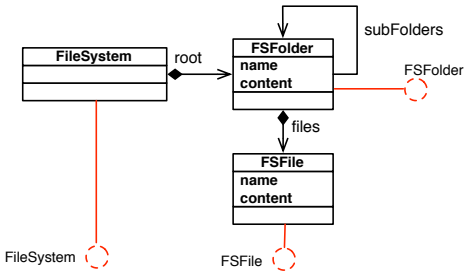

Figure 5: Core classes model a file system.

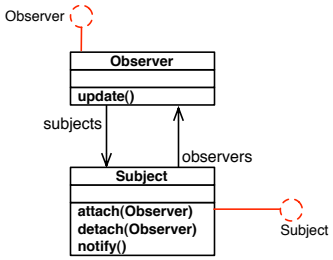

Figure 6: Aspect classes model the observer pattern.

eling language Ecore [25] using a dedicated composition system.

Consider the example below where the structure of a file system (see Figure 5) and an Observer pattern [30] (see Figure 6) are modeled in Ecore. The `Subject` and `Observer` classes are aspect classes. We can weave them into the file-system model to add subject–observer roles to it. Both models conform to an extended Ecore metamodel, the *ReuseEcore* metamodel.

Elements that are variation points are visualized by attached circles in the figures, where the variation points `FileSystem`, `FSFile`, and `FSFolder` are hooks and the variation points `Observer` and `Subject` are anchors that can be accessed as an extension to a hook or a replacement for a slot.

In the composition system we realize, `EClass`es can act as *aspect classes*, modeling a certain aspect that can be inserted into other `EClass`es—*core classes*. Aspects are woven into a core class by taking the following steps: 1) copy all `EStructuralFeatures` and all `EOperations` contained in the aspect class into the core class; 2) if an aspect class references another aspect class (e.g., as the `eType` of an `EReference`), the reference is changed into a reference to the core class that aspect class is woven into within this composition step; 3) if there is more than one such core class, multiple copies of the same reference, each pointing at a different core class, are bound.

The aim is to hide these details from the users of the developed LWDCS. Thus, a user of the particular LWDCS only has to define which aspect classes are woven into which core classes.

The weaving can be encapsulated in a dedicated composition operator (see Section 5). To call the composition operator, a small dedicated and refined composition language (the *ecoreweaving language*) is defined (cf. Figure 4). Notice that this differs from the Xcerpt example where the composition language is an integral part of the reuse language. Listing 10 shows a composition program specifying the weaving.

```
//find the metamodel that defines the language reuse ecore                1
language recore : http://www.reuseware.org/ReuseEcore;
                                                                            3
//load the core
fragmentlist recore.EPackage coreFs = /FileSystem.recore;                   5

//load the observer aspect                                                  7
fragmentlist recore.EClass observer = /ObserverAspect.recore;
                                                                            9
//weave using the EcoreWeaving composition operator
weave (                                                                     11
```

```
  coreFs[name="filesystem"].eClassifiers[name="FileSystem"] <--
      observer[name="Observer"],                                    13
  coreFs[name="filesystem"].eClassifiers[name="FS.*"]       <--
      observer[name="Subject"]                                      15
);
                                                                    17
//print back the composition result
print coreFs to /FSObs2.ecore;                                      19
```

Listing 10: A composition program written in the dedicated *ecoreweaving* composition language.

First, the two model fragments from Figures 5 and 6, modeled in the extended Ecore language *ReuseEcore*, are loaded (see Lines 5 and 8). Then the dedicated composition operator (`weave`) is called, accepting pairs of aspect classes and core class lists as arguments (see Lines 11–16). Note that the arguments are fragments contained in the loaded fragments (e.g., core classes in the files system package). They can be addressed because they are variation points (as illustrated in Figure 5 and 6). Finally, the composition result is printed to a file (Line 19).

This example hinted at how an LWDCS can be utilized to introduce new reuse abstractions in a modeling language. We will continue to investigate other abstractions for modeling languages and continue to refine the current methods to make these techniques directly available to modelers.

5 Complex Composition Operators and Refined Component Models

As argued, the composition technique of ISC is very flexible and this characteristic is one of its advantages. The flexibility is achieved through the provision of primitive composition operators. An important disadvantage of these, as also mentioned in Section 2.1, is that they are only able to express low-level composition steps. To fully realize a useful abstraction, for example the module system presented in Section 4.1, several such low-level composition steps must be joined together into an atomic unit. This encapsulation of a set of low-level operators allows for reusing high-level composition steps—as high-level *complex composition operators*. Figure 7 illustrates schematically how the complex operators are defined in terms of the available primitive operators and how different complex operators are defined for different languages and purposes. It shows two operators explained in Section 4.1—*to-module* and *import*—and how they can be implemented using the generic composition operators provided by ISC.

For the realization of module encapsulation in the module system, possible dependencies between rules must be controlled. That is, a rule in a module should not depend on a rule of the importing program in the composition result unless explicitly desired. This separation of rules is realized by the notion of *stores*. Data constructed by rules in a specific module is re-directed into its store. Thus, to access data constructed by a module, its store needs to be accessed. Access to different stores is granted via special-purpose constructs (*to-module* and *in-module*) realized as complex composition operators. Listing 11 shows the definition of the *to-module* composition operator. The

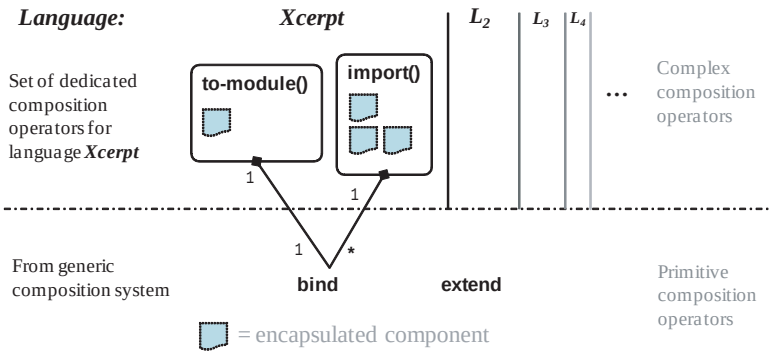

Figure 7: Dedicated complex composition operators are defined in terms of the generic primitive composition operators provided by ISC.

construct is intended to be used in programs importing a module to construct data that will be read and accessed by rules in the module (see Section 4.1). One thing to notice about complex composition operators is that they may not only encapsulate a sequence of primitive operators, but also fragments. That is, some composition operators require internal fragments for the implementation of the (abstraction) construct it is realizing.

The first statement in the operator definition declares such an operator-internal fragment (see Lines 3–7) containing a construct term with a variation point (`cTerm`). This fragment is used to re-direct the constructed data to the appropriate store. The construct term (in the composition program) on which the operator is executed is passed as a parameter (`constructTerm`) to the operator and is bound to the variation point (`cTerm`) using the primitive operator *bind*. Finally, the resulting construct term is returned, replacing the location where the operator was executed with the newly produced construct term. This is an example of an inline composition as discussed in Section 4. As such, the constructed data is properly re-directed to the appropriate store. The same technique can be used for re-directing queries in the realization of the *in-module* construct. The *import* construct in turn re-directs all queries and data constructions in the imported module. The details are however not further discussed here.

```
define composer modularxcerpt.ToModule(moduleName, constructTerm) {          1

    fragmentlist xcerpt.ConstructTerm ctWrapper =                            3
       'store [ modul ["' + ->moduleName +  '"],
                visibility [ "public" ],                                     5
                <<cTerm>>
              ]'.mxcerpt;                                                    7

    bind cTerm on ctWrapper with constructTerm;                              9

    return ctWrapper;                                                        11
}
```

Listing 11: Definition of the *to-module* composer used in the Xcerpt module system.

From a usage perspective, an Xcerpt module can be seen as a black-box with an input rule and an output rule. As such, the *component interface* is well understood

by users of the LWDCS realizing the module system. The *composition interface* of a module component, however, cannot be considered as a black-box, as previously argued. As explained, the realization of the module system transforms query and construct terms to enforce proper module encapsulation. The possibility of transforming these rule parts must be reflected in the relevant component model.

On a practical level, it is not desirable for module programmers to know the specifics of how the module system is implemented. Rather, module programmers should only have to define reusable modules and declare which rules should be accessible to importing programs and which rules should be fully encapsulated, that is, not accessible at run-time. Thus, one should not require the module programmers to open up their modules in a very composition-specific way.

Using the grammar extension techniques presented in Section 3, one could make the constructs `ConstructTerm` and `QueryTerm` variable and thus produce a candidate component model for the realization of the system. However, such a component model would only allow for the declaration of slots as alternatives for concrete query and construct terms. Such a component model would disallow complex composition operators transforming rules in the required way. To achieve a balance between ease for users and correctly accessed components, we create component models in a more refined way than was so-far described. This is a direct result of the introduction of the possibility to define complex composition operators. Notice that this does not invalidate the previously described component models, but builds on-top and refines them.

When an abstract construct C of some language is made variable, the refined method transforms the grammar of the language such that either i) a slot (with concrete syntax) is allowed to be defined in its place, or ii) a *default* concrete instance of C can directly be programmed into its place. In any respect, in the abstract syntax description of the reuse grammar, C is referencing the upper-level component model and is specified to be a variation point. Thus, the component model captures both the possibility to define explicit variation points (slots) and to directly provide an instance belonging to the same syntactical category as C. What is important is that the relevant location in a fragment is accessible wrt. the component model—part of the composition interface of the fragment.

6 Composition Framework Implementation

This section describes the *Reuseware Composition Framework* that implements the concepts described above. The framework can be used to develop LWDCSs realizing abstractions for different languages. For example, the above- described module system for Xcerpt was developed using the framework. The framework itself can be used independently of other technology. To simplify usage, we have also built a frontend in the form of a set of plug-ins for the Eclipse Platform [31].

The framework consists of two distinct parts. The first part provides tooling for defining new LWDCSs, including facilities to extend languages and to define complex composition operators. The second part provides support for using so-defined LWDCSs; that is, support for defining fragments and executing compositions. In the following we go through the process of defining and using a LWDCS realizing the

module system for Xcerpt from Section 4.1.

6.1 Generating a LWDCS

As a basis for developing a new LWDCS, we require a description of the core language—in our case Xcerpt. The Eclipse Modeling Framework is used to generate a Java code representation of the Ecore metamodel. Utilizing the ANTLR tool set [32], a parser and a printer are generated based on the concrete syntax description.

The abstract and concrete syntax grammars of Xcerpt can be derived by separating concrete from abstract elements in the (partial) grammar from Listing 2. Additionally, each reference to a production rule in the abstract syntax grammar is tagged with a *role name* (`rolename : NonTerminal`). These names are then used in the definition of the concrete syntax.[11] The production rule for `GoalQueryRule` (Line 6 in Listing 2), for instance, is split into an abstract and a concrete form. The abstract syntax production rule looks as follows:

```
GoalQueryRule = goal:ConstructTerm, query:QueryTerm?;
```

This abstract syntax rule is then annotated with its concrete syntax using the notion of role names in the following manner:

```
GoalQueryRule ::= "GOAL" goal ("FROM" query)?  "END";
```

After the grammars of the original language have been defined in the tool, they can be extended for reuse. Currently, such extensions are done manually, but it would also be possible to generate them (semi-)automatically (see Section 8 for a discussion). The newly introduced production rules are annotated with concepts from the upper-level component model such that they can be identified during grammar-to-metamodel transformation.

Figure 8 visualizes the extended grammars of Xcerpt that use annotations to identify non-terminals that describe variation points. Such annotations are done in the abstract syntax grammar by using the non-terminal name followed by **==>** and the name of a metaclass from the upper-level component model, such as `componentmodel.Slot` or `componentmodel.Hook` (see Section 3.1).

These new production rules are placed in a separate file. The complete extended grammar then consists of the original grammar as modified by these additional production rules. Non-terminals defined elsewhere can be referred to using the `Language.NonTerminal` notation. Thus a separation between core language and reuse extension is reflected on the grammar files.

Note, that a concrete syntax definition has to be provided for the reuse language. However, such definitions do not have to differentiate between core and extended production rules, since the parsers are generated from the complete reuse language definition.

Based on these specification the LWDCS can now be generated. Figure 8 shows the result of the automatic generation process—labeled “generated code”.

[11] Assigning role names also improves the result of the grammar-to-metamodel transformation, since they can be transformed to reference names.

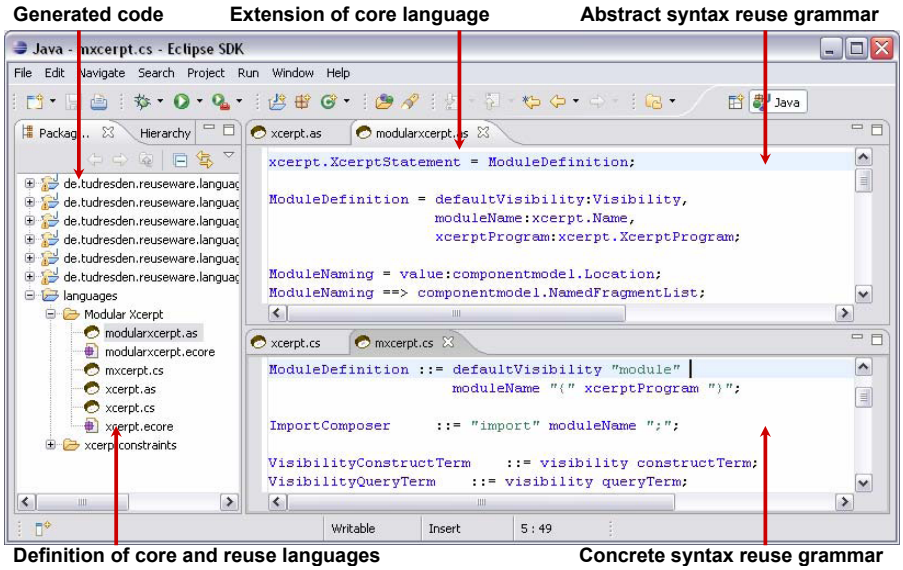

Figure 8: Grammar of *Modular Xcerpt* – LWDCS development in Reuseware.

6.2 Executing composition

Once a LWDCS is generated, it can be deployed and activated inside the Eclipse Platform. In our case, fragments written in the core language Xcerpt and its reuse extension are now understood by the tooling and variation points inside fragments can be recognized and addressed for composition. Compositions are executed by merging abstract syntax trees of fragments and using a generated printer to obtain a concrete program as result. Since grammatical types are represented in the language metamodel and in the generated code, type-safe composition is ensured.

Figure 9 shows the *Reuseware* environment where fragments can be defined and composition programs can be executed. When working in a composition environment, a *Reuseware Project* can be created providing support for the creation of folders where fragments can be stored, composition programs loaded and composition results printed. The discussed Xcerpt files (modules, programs etc.) would be put in the relevant folders.

7 Related Work

The *Mjølner System* and the *Beta language* [33] were the first to introduce the concept of slots. In Beta, any programming construct can be replaced by a slot typed with the non-terminal corresponding to that construct. Beta also supports a notion of inheritance of grammar types. Binding of slots happens when the name of a fragment and the name of a slot in the same project match. Our approach extends the Beta approach in two ways:

1. We introduce additional types of variation points, like hooks, which can be ex-

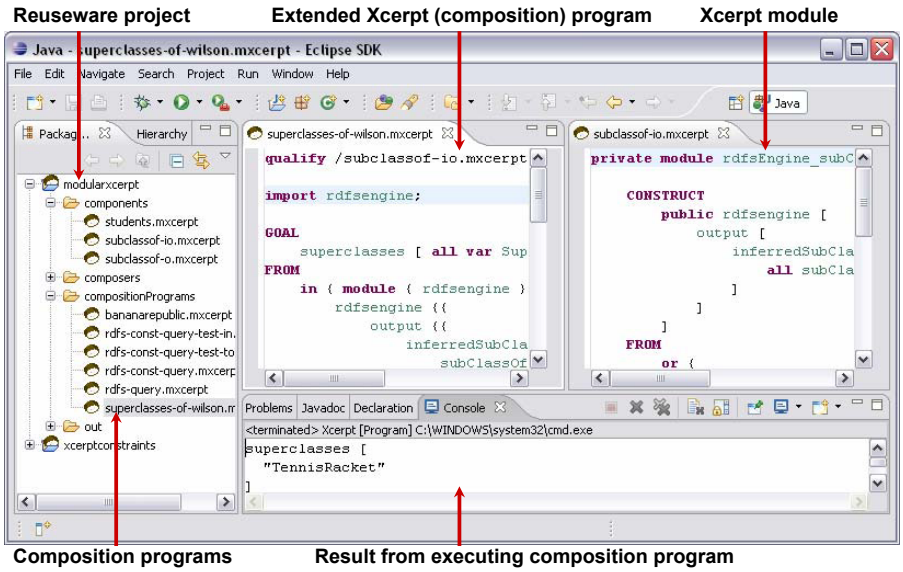

Figure 9: Xcerpt composition environment realized in Reuseware.

tended multiple times. Also, we make explicit the actual composition operators, so that binding a slot with a fragment is an explicit operation rather than implicitly matching by name.

2. We extend the concept to any language that can be described by a context-free grammar. Different from Beta, our tool allows any language to be extended with a composition system.

The *Software COMPOsition SysTem* (COMPOST) [34] is a predecessor of our current system, which introduced many of the concepts available in our approach, but was limited to Java and XML. For each new language that should be supported by COMPOST a large amount of implementation work is required. Additionally, the definition of dedicated composition languages is not supported. There is only one composition language, namely Java.

Our notion of fragment components is comparable to the notion of *syntactic units* presented in [35]. Syntactic units are arranged in *syntactic unit trees* that can be likened to composition programs. In this approach, so-called extension spots can be defined as alternatives for any fragment of code derivable from a non-terminal. Compared to our approach, there is no formalization of language extensions which allows for tailored extension of a language (to only allow the desired amount of variability) and generation of language specific tooling.

In [36], Gray and Roychoudhury present a technique for constructing aspect weavers for arbitrary languages. They define an aspect weaving language (called *Aspect Domain*) which can be used to define weavings for different languages. They argue that a common superset of weaving operations can be applied to arbitrary languages, while certain languages require specific extensions. The weaving language is comparable

to our composition language. The language-independent weaving operations can be compared with primitive composition operators and language-specific operations with complex composition operators. The major differences is that Gray and Roychoudhury do not extend languages, because their components (that is, *core* and *advice* artifacts) only have implicit composition interfaces—which is reasonable, since they focus on legacy systems written in existing languages—while we focus on DSLs under development.

Gray and Roychoudhury explain in detail how their implementation is built on top of a transformation engine (which is the Design Maintenance System [37]). They show how this raises the level of abstraction and eases aspect development and weaving. This is also true for the Reuseware implementation: It is build on top of the EMF and ANTLR frameworks, which can also be seen as transformation engines, and raises the level of abstraction wrt. component-based development. Gray and Roychoudhury state that their concepts could be implemented on top of other transformation engines. While not explicitly discussed in detail, the same holds for the Reuseware concepts: they could likewise be implemented on top of program transformation systems like TXL [38], ASF+SDF [39], or others. However, we chose an EMF-based implementation to achieve good Eclipse integration and good support for metamodel-based languages.

8 Conclusions and Outlook

In this paper we presented a formalism for realizing Invasive Software Composition [10], a gray-box approach to composition, for arbitrary languages. We showed how languages can be augmented with reuse abstractions via composition, where the introduced formalism plays a central role. The specific reuse concepts can be realized via special-purpose composition operators. These form an important part of specialized composition environments—dedicated composition system (LWDCS). To show how this can be done in practice we augmented the Web query language Xcerpt with the possibility to define reusable *modules*. In Section 6 we presented an Eclipse-based tool that provides support for the introduced composition concepts and allows for the development of LWDCSs.

The work presented in this paper only supports composition at a syntactic level. That is, composition results may be semantically invalid. We work toward reusing existing techniques and tools to ensure semantically valid compositions. We group these possibilities in two main categories: 1) *explicit contracts* and 2) *implicit verifications*. Explicit contracts allow programmers to restrict how their components may be used. Implicit verification can be employed in two ways. First, by applying existing tools to check composition results for general errors. For example, applying an ontology reasoner to check a composed ontology for inconsistencies. Second, one can apply the same tools for specialized reasons in specific LWDCS. An example would be to check the correct usage of Xcerpt modules (see Section 4.1) and report only possible interface errors, even though other unrelated errors might be present. In the Xcerpt example, this means that the structure of data provided to a module must match the structure of the date expected by the module.

While our tool can automatically generate an invasive component model for different languages, it is currently not able to generate a refined component model as needed for most LWDCS (see Section 5). Currently, these refinements have to be performed manually by the developer of the LWDCS. However, as was noted in Section 4, there is a close correspondence between complex composition operators and component models in a LWDCS (see Figure 4). Hence, our next step is to derive refined component models from a set of composition operator definitions. Being able to generate large parts of a LWDCS is of great importance for the applicability of our approach.

As we discussed in Section 4.2, it is important to support the extension of metamodel-based languages, for example, Ecore. While our tooling already supports metamodel-based extensions, finding the precise limitations of this approach in this respect is part of future work.

This leads to the question of the overall limitations of the presented approach. The limitations must be addressed on two different levels. First, wrt. the languages being targeted, and second, the specific reuse and abstraction construct being realized for any given languages. The main requirement on the language is that it must be context-free. For a textual language it means that its grammar is context-free and for a metamodel-based language that its metamodel has a tree-like structure or that a skeleton tree can be overlaid onto any metamodel instance. Another general limitation of the approach, as mentioned above, is the restriction to purely syntactical compositions. We must also consider what kinds of components can be realized for a particular language. As components are compiled to the underlying core language, the desired component properties must be enforceable in that language. For Xcerpt, for example, a desired component property is encapsulation, which turned out to be enforceable using the concept of *stores*. For other languages and component types, certain component properties might not be satisfiable. As more languages and component types are experimented with, we hope to gain experience in this respect.

Acknowledgment

This research has been co-funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. http://rewerse.net), as well as through the 6th Framework Programme project Modelplex contract number 034081 (cf. http://www.modelplex.org) and by the German Ministry of Education and Research (BMBF) within the project feasiPLe (cf. http://www.feasiple.de).

References

[1] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.

[2] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. OWL web ontology language semantics and abstract syntax. W3C Recommendation, February 2004. Available at http://www.w3.org/TR/owl-semantics/. Accessed May 2007.

[3] Dan Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation, February 2004. Available at http://www.w3.org/TR/rdf-schema/. Accessed May 2007.

[4] Ian Horrocks and Peter F. Patel-Schneider. A proposal for an OWL rules language. In *Proc. of the 13th International World Wide Web Conference (WWW 2004)*, pages 723–731. ACM, 2004.

[5] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML query language. W3C Recommendation, January 2007. Available at http://www.w3.org/TR/xquery. Accessed May 2007.

[6] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. W3C Working Draft, March 2007. Available at http://www.w3.org/TR/rdf-sparql-query/. Accessed May 2007.

[7] François Bry and Sebastian Schaffert. The XML query language Xcerpt: Design principles, examples, and semantics. In *Revised Papers from the NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems*, pages 295–310. Springer-Verlag, London, UK, 2003.

[8] Object Management Group. Unified Modeling Language: Superstructure version 2.0. OMG Document, August 2005. Available at http://www.omg.org/docs/formal/05-07-04.pdf. Accessed May 2007.

[9] Oscar Nierstrasz and Theo Dirk Meijler. Research directions in software composition. *ACM Computing Surveys*, 27(2):262–264, June 1995.

[10] Uwe Aßmann. *Invasive Software Composition*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[11] William Harrison, Harold Ossher, and Peri Tarr. Asymmetrically vs. symmetrically organized paradigms for software composition. Technical report, IBM, 2002.

[12] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina V. Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proc. of the 11th European Conf. on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *LNCS*, pages 220–242, Heidelberg, 1997. Springer.

[13] Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.

[14] Jakob Henriksson, Uwe Aßmann, Florian Heidenreich, Jendrik Johannes, and Steffen Zschaler. How dark should a component black box be? The Reuseware Answer. *Proc. of the 12th International Workshop on Component-Oriented*

Programming (WCOP) co-located with 21st European Conf. on Object-Oriented Programming (ECOOP'07) (to appear), 2007.

[15] Jakob Henriksson, Jendrik Johannes, Steffen Zschaler, and Uwe Aßmann. Reuseware – adding modularity to your language of choice. *Proc. of TOOLS EUROPE 2007: Special Issue of the Journal of Object Technology (to appear)*, 2007.

[16] Object Management Group. Meta Object Facilities (MOF) 2.0 Query / View / Transformation Specification. OMG Document, November 2005. Available at http://www.omg.org/cgi-bin/apps/doc?ptc/05-11-01.pdf. Accessed May 2007.

[17] Atlas Project Team. Atlas Transformation Language. Available at http://www.eclipse.org/m2m/atl/. Accessed May 2007.

[18] James Clark. XSL transformations (XSLT). W3C Recommendation, November 1999. Available at http://www.w3.org/TR/xslt. Accessed May 2007.

[19] Sebastian Schaffert, Francois Bry, and Tim Fuche. Simulation unification. Technical Report IST506779/Munich/I4-D5/D/PU/a1, Institute for Informatics, University of Munich, 2005.

[20] Dexter C. Kozen. *Automata and Computability*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[21] International Organization for Standardization. *ISO/IEC 14977:1996: Information technology — Syntactic metalanguage — Extended BNF*. International Organization for Standardization, Geneva, Switzerland, 1996.

[22] Francois Bry, Tim Fuche, and Sebastian Schaffert. Initial draft of a language syntax. Technical Report IST506779/Munich/I4-D6/D/PU/a1, Institute for Informatics, University of Munich, 2006.

[23] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering Methodology*, 14(3):331–380, 2005.

[24] Object Management Group. MetaObject Facility (MOF) specification version 2.0. OMG Document, January 2006. Available at http://www.omg.org/cgi-bin/doc?formal/2006-01-01. Accessed May 2007.

[25] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[26] Marcus Alanen and Ivan Porres. A relation between context-free grammars and meta object facility metamodels. Technical Report 606, TUCS - Turku Centre for Computer Science, Turku, Finland, March 2004.

[27] Uwe Aßmann, Sacha Berger, François Bry, Tim Furche, Jakob Henriksson, and Jendrik Johannes. Modular web queries—from rules to stores. *3rd International Workshop On Scalable Semantic Web Knowledge Base Systems (SSWS'07) (to appear). Vilamoura, Algarve, Portugal, Nov 27, 2007*, 2007.

[28] Adrian Colyer, Andy Clement, George Harley, and Matthew Webster. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools (The Eclipse Series)*. Addison-Wesley Professional, 2004.

[29] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban. AspectC++: an AOP Extension for C++. *Software Developer's Journal*, 5:68–76, 2005.

[30] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1994.

[31] The Eclipse Foundation. Eclipse platform technical overview. Available at http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html. Accessed May 2007.

[32] Terence Parr. ANTLR — ANother Tool for Language Recognition — parser generator. Available at http://www.antlr.org. Accessed May 2007.

[33] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, June 1993.

[34] The COMPOST Consortium. The COMPOST system. Available at http://www.the-compost-system.org. Accessed May 2007.

[35] Marek Majkut and Bogdan Franczyk. Generation of implementations for the model driven architecture with syntactic unit trees. In *Proceedings of 2nd OOPSLA Workshop Generative Techniques in the context of MDA*, October 2003.

[36] Jeff Gray and Suman Roychoudhury. A technique for constructing aspect weavers using a program transformation engine. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 36–45, New York, NY, USA, 2004. ACM Press.

[37] Ira D. Baxter. Dms: program transformations for practical scalable software evolution. In *IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution*, pages 48–51, New York, NY, USA, 2002. ACM Press.

[38] James R. Cordy. The txl source transformation language. *Sci. Comput. Program.*, 61(3):190–210, 2006.

[39] Mark G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the asf+sdf compiler. *ACM Trans. Program. Lang. Syst.*, 24(4):334–368, 2002.