

Reuseware – Adding Modularity to Your Language of Choice

Jakob Henriksson Fakultät für Informatik, Technische Universität Dresden

Jendrik Johannes Fakultät für Informatik, Technische Universität Dresden

Steffen Zschaler Fakultät für Informatik, Technische Universität Dresden

Uwe Aßmann Fakultät für Informatik, Technische Universität Dresden

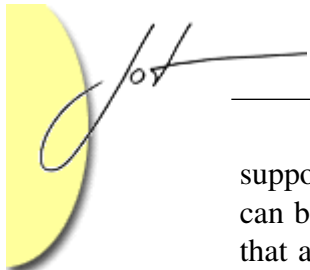
The trend towards domain-specific languages leads to an ever-growing plethora of highly specialized languages. Developers of such languages focus on their specific domains rather than on technical challenges of language design. Generic features of languages are rarely included in special-purpose languages. One very important feature is modularization, the ability to formulate partial programs in separate entities, composable into a complete program in a defined manner. This paper presents a generic approach for adding modularity to arbitrary languages, discussing the underlying concepts and presenting the Reuseware Composition Framework. We walk through an example based on Xcerpt, a Semantic Web query language.

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 one example of this kind of languages. The Semantic Web [3] is another area, in which lots of languages have been defined to precisely capture meaning and enable accurate description of data found on the web. Examples of Semantic Web languages include ontology languages (e.g. OWL [24], RDF(S) [5], SWRL [11]) and query languages (e.g. XQuery [4], SPARQL [25], Xcerpt [7]). Another important area where many languages exist is modeling, and many DSLs have appeared around the Unified Modeling Language (UML) [20].

These languages have been carefully designed and are very capable in their domains of operation. The language designers focus on the domain-specific concepts and issues the languages are meant to support. However, a lot remains to be wished for regarding some more technical issues of language design. For example, many of these languages are lacking good concepts for modularization, and, thus, have insufficient support for reuse. It is well-known in the software engineering community that there are many benefits to be harvested from creating programs based on reusable components. This method is considered a vital part of large mature systems [19]. Yet, providing good support for modularity and reuse can be a quite complex task in its own right. Therefore, it is often neglected when new languages are being constructed.

To address this problem one could re-design the individual languages such that they



support modularity and composition-based thinking and development. As mentioned, this can be a quite daunting task. On the other hand, modularization has a lot of properties that are independent of concrete languages. Hence, one can consider providing modularization support on a language-independent level, without initially tampering with the specifics of each individual language. This has the advantage that the same composition technique can be reused for any language. More importantly, one can create a general tool framework for enabling component-based development for a number of languages lacking such capabilities.

A foundation of our work is Invasive Software Composition (ISC) [2], a composition technique that is attractive for two reasons:

1. It is flexible wrt. the granularity of components and symmetry of composition [10] and can, thus, capture and simulate many well-known composition techniques (e.g. aspects [13] and hyper spaces [22]).
2. It operates at source-code level and, thus, is suitable also for interpreted, declarative, or descriptive (i.e. non-operational) languages—for example, for the languages of the Semantic Web or for modeling languages, such as UML.

So far, the concepts of modularity made available through ISC have been manually implemented for Java and XML [27]. These concepts, however, can be understood independently of a concrete language. The contribution of this paper is twofold:

1. We give a formal explanation of how ISC, and its concepts for modularity, can be made available for arbitrary languages.
2. We present a framework and tool¹ that embodies these concepts. It can generate language extensions for modularization support for an arbitrary language, given its description. Furthermore, this tool also allows to address language-specific issues and requirements.

Hence, this paper can be viewed as a contribution to the vision of *grammarware engineering* [14], the systematic development and maintenance of grammar-based applications.

As a running example, the paper shows an extension of the XML query and transformation language Xcerpt [7] with concepts for modularization.

The remainder of this paper is structured as follows. Section 2 deals with some preliminaries, introducing both ISC and Xcerpt in more detail. In Section 3 we discuss the formal concepts involved in a language-independent rendering of ISC followed by a presentation of the tool in Section 4. While we give formal definitions for all our concepts, we also use a running Xcerpt example to illustrate the effect of these definitions. Section 5 references related work, and Section 6 concludes and presents directions for future work.

¹The Reuseware Composition Framework, available from <http://www.reuseware.org>



2 PRELIMINARIES

Before we describe our formalism in Section 3 for extending an existing language for modularity, we give a brief introduction to ISC. However, rather than explaining the details of ISC, which is better learned by studying [2], we motivate why we build our work upon, and extend, ISC. As we apply and demonstrate our techniques on the Web query language Xcerpt, we introduce the language in the second part of this section.

Invasive Software Composition

ISC is a grey-box composition approach where components—or fragments components²—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. The replacement of a hook with some existing component constitutes the basic composition technique of ISC. The benefit of this composition technique is that it is very general and is realizable for any language used to author the components. Another attractive quality in grey-box approaches is the flexibility of components wrt. to granularity and symmetry, properties that differentiate many known composition techniques [10]. Thus, ISC can realize and combine techniques that rely on well-defined component interfaces [2], for example, aspect-oriented programming [13], hyper-space programming [22] and view-based programming [2].

Grey-box approaches stand in contrast to traditional *black-box* approaches where users of components solely rely on expected inputs and outputs of components. For modeling and Semantic-Web languages, traditional black-box approaches are not applicable, because there is no notion of a dynamic execution of components. ISC works on component source code and is, thus, easily applicable to composition in the modeling and Semantic Web domains. Our contribution in this paper is a formalization of the notion of ISC fragments. The important point, however, is that this formalization is not bound to a particular language. Thus, we describe a formalism where arbitrary languages can be extended to take advantage of the possibilities of using ISC's grey-box techniques.

Xcerpt

Xcerpt is an XML query and transformation language. In contrast to similar languages like XQuery [4] and XSLT [9], 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*

²Simply *fragments* when it is clear from the context what is meant.

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 and there is a one-to-one correspondence between the two notions. Xcerpt data terms use a square bracket notation, e.g. the data term `book [title ['White Mughals']]` corresponds to the XML snippet `<book><title>White Mughals</title></book>`. 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 technique³ to match data terms. 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. E.g. 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 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

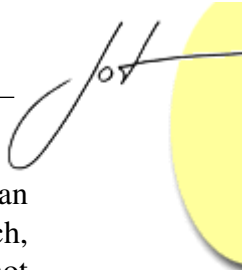
CONSTRUCT                                7
  book [ title [ "White_Mughals" ], author [ "William_Dalrymple" ] ]
END                                     9

```

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 view into Xcerpt, please consult [7].

³*simulation unification*, for details of this technique, please refer to [26]



The reuse abstraction available directly 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 support to reuse query terms or any other syntactical entity supported by the language. In the following sections we propose how this situation can be remedied for any language defined by a context-free grammar. It should be noted that while we use examples from Xcerpt, the techniques is in no way limited to this particular language.

3 FORMALISM

Before defining our notion of fragment components 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. We will then base our explanations on languages defined using context-free grammars. Fragment components and ISC can also be used for languages based on metamodeling, but in this paper we focus on grammar-based languages to simplify our explanations.

Formally, a context-free grammar (CFG) is a 4-tuple [15]:

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

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;	
ConstructQueryRule	= "CONSTRUCT", ConstructTerm, ("FROM", QueryTerm)?, "END";	3
GoalQueryRule	= "GOAL", ConstructTerm, ("FROM", QueryTerm)?, "END";	5
QueryTerm	= StructuredQt ...	7
ConstructTerm	= ...	

Listing 2: A selection of the production rules for Xcerpt

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 (based on a grammar given in [6]). 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. The symbols ? and | have the standard meaning as used in EBNF [12].

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 and starting with the start symbol S . We can, therefore, say that P is of (*grammatical*) type S .

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 . 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. Note that 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.

The program in Listing 3 queries a bibliography database `biblio.xml` and extracts titles and authors from it and 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. Therefore its set of grammatical types includes these non-terminals.

```

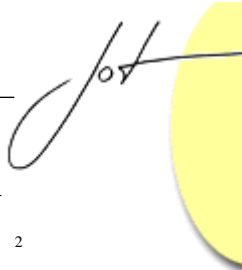
GOAL
  results [ all result [
    var Title, all var Author
  ] ]
FROM
  in { resource { "file:biblio.xml", "xml" },
    bib [[ book [
      var Title -> title [[ ]],
      authors [[
        var Author -> author [[ ] ]
      ] ]
    ] ] ]
}
END

```

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: 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. to 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. to the original Xcerpt grammar where the start symbol has been changed to QueryTerm.



```

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

```

Listing 4: A partial Xcerpt program: a query term

Second, partiality of a program can also come from within a specific sub-part as it was defined above. Thus, for a partial program it is not sufficient to only use a different start symbol for the grammar. 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. 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 « and » to markup the variation point and myVarPoint is the identifier for the specific variation point.

```

GOAL
  results [ all result [
    var Title, all var Author
  ] ]
FROM
  <<myVarPoint : QueryTerm>>
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_{r_I} 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 of \mathcal{L} . It is a generalising grammar transformation in the sense of [14]. 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 the rule QueryTerm is replaced by a choice. Such a choice allows to specify a variation point — $v(\text{QueryTerm}, I)$ — as an alternative to a concrete query term.

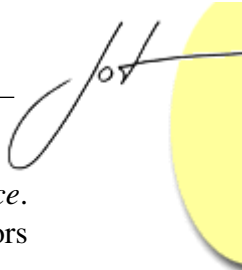
XcerptProgram	= XcerptStatement+;	1
XcerptStatement	= GoalQueryRule ConstructQueryRule;	
ConstructQueryRule	= "CONSTRUCT", ConstructTerm, ("FROM", (QueryTerm $v(\text{QueryTerm}, I)$))?, "END";	3 5
GoalQueryRule	= "GOAL", ConstructTerm, ("FROM", (QueryTerm $v(\text{QueryTerm}, I)$))?, "END";	7
$v(\text{QueryTerm}, I)$	= "<<", I, ":", "QueryTerm", ">>";	9
QueryTerm	= StructuredQt ...	11
ConstructTerm	= ...	

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

There are two basic approaches to derive 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 , or *universally variable*.
2. Introduce variation points for a well-chosen subset of non-terminals in V_n of G only (i.e. $V_v \subset V_n$). In this case, we call G_I a *tailored extension* of G .

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 *invasive composition interface*. Based on their composition interfaces, fragments can be connected using special operators called *composition operators*.

Definition 5 (Primitive composition operators). A primitive composition operator is an operation that takes two fragments F_1 and F_2 , and the identifier of a variation point v in F_1 and produces a new fragment, which is equal to F_1 , but with the variation point transformed by F_2 .

Depending on the type of compositions allowed on them, we distinguish two basic types of variation points (as well as the corresponding types of primitive composition operators):

1. *Slots* can be used for parametrization of a fragment. They function as place holders for a single fragment. Once a slot has been *bound* (i.e., the composition operator `bind()` has been applied to the slot), it cannot be bound again.
2. *Hooks* can be used for extension of fragments. A hook serves as an extension point where suitable fragments can be added repeatedly, if necessary, by applying the `extend()` composition operator to the hook. That is, after each call of `extend()` the hook being operated on remains in place to be used again.

Composition operators are type-safe operators in the sense that they enforce that the type of the variation point and the type of the fragment to be bound to the slot, or added to the extension point, match. Definition 6 specifies what matching of grammatical types means. It should be clear that the types involved are derived from the underlying core grammar and its set of non-terminals.

Definition 6 (Type safety). Let G be a context-free core grammar, F_1 and F_2 fragments valid wrt. $G_I = \tau(G)$, $v(v_n, I)$ a variation point in F_1 , and c a primitive composition operator. Let $GT_2 \subseteq V_n$ be the set of grammatical types of F_2 . Then c can be applied to F_1 , F_2 , and v , iff $v_n \in GT_2$; that is, if the grammatical types of the variation point and the fragment to be composed match.

By enforcing type safe composition, it is guaranteed that the result from executing a composition operator will always be a valid partial program wrt. the underlying core grammar. Thus, syntactical errors in the potential composition result will already be caught at composition time and reported to the user. In case of a type error, the composition will not be executed.

For example, the grammatical types of the query term in Listing 4 are $\{\text{QueryTerm}, \text{StructuredQt}\}$. Binding the variation point in Listing 5 with the fragment in Listing 4 using the `bind()` primitive composition operator is type safe. This is so, because the type of the variation point belongs to the set of grammatical types of the fragment being bound ($\text{QueryTerm} \in \{\text{QueryTerm}, \text{StructuredQt}\}$). Trying to bind the fragment in Listing 3 to the same variation point will result in a type error and the execution will not proceed (since $\text{QueryTerm} \notin \{\text{XcerptProgram}, \text{XcerptStatement}, \text{GoalQueryRule}\}$).

Additionally to the concept of primitive composition operators, we introduce the notion of *complex composition operators*. A complex composition operator groups a number of primitive operators, intended to be executed as an atomic unit for some specific task. As such, the complex operator can address several variation points acting together and apply the primitive composition operators on them in sequence. We will in the following demonstrate the use of such a complex composition operator implementing a module system for 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. XSB⁴. Here we intend to solve this issue using our composition framework. An Xcerpt module in this setting is an Xcerpt program with slots, i.e. an Xcerpt fragment. The slots can be used to configure modules for some specific purpose, e.g. to control the information flow, as we will show below.

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 *subclass-of* hierarchy. Ontology reasoners are often employed to infer implicit information contained in such ontologies, e.g. to compute the transitive closure of the 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. The second rule is used to extract the explicit subclass-of information from the ontological data, while the first rule infers the implicit information.

```

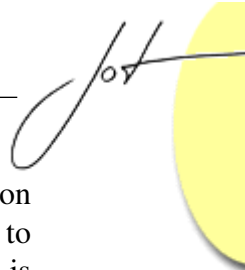
CONSTRUCT
  inferredSubClassOf [
    all subClassOf [ var Subclass, var Superclass ]
  ]
FROM
  or {
    declsubclassof [ var Subclass, var Superclass ],
    and {
      declsubclassof [ var Subclass, var Z ],
      declsubclassof [ var Z, var Superclass ]
    }
  }
END

CONSTRUCT
  declsubclassof [ var Subclass, var Superclass ]
FROM
  <<rootNode>> [[
    Class {
      id { var Subclass },
      subClassOf {
        about { var Superclass }
      }
    }
  ]]
END

```

Listing 7: An Xcerpt module in the file /subClassOf.mxcerpt

⁴<http://xsb.sourceforge.net>



Since the author of the reusable module does not necessarily know how the data on which the rules will operate will be structured, a slot has been declared to be able to configure this. A possible extended Xcerpt program using the module defined above is shown in Listing 8. The program uses the module to construct a result consisting of all the existing subclass-of relationships of an ontology. For simplicity, the input data (i.e. the ontology) is directly constructed inside the program (Lines 11–19), but could also be given as a view provided by an additional construct rule querying a specific ontology document and format (e.g. OWL [24]). The given ontology describes sports articles and their subclass-of relationships.

```

IMPORT
/subClassOf.mxcerpt [ bind(rootNode, 'owl') ]      2
END                                              4

GOAL
  result [ all var X ]                            6
FROM
  var X -> inferredSubClassOf [ [ ] ]              8
END                                              10

CONSTRUCT
  owl [                                          12
    Class [ id [ "SportsEquipment" ] ],
    Class [ id [ "TennisRacket" ],
            subClassOf [ about [ "SportsEquipment" ] ] ],
    Class [ id [ "WilsonTennisRacket" ],
            subClassOf [ about [ "TennisRacket" ] ] ]
  ]                                              18
END

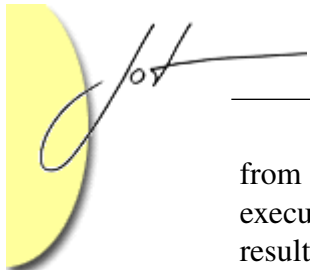
```

Listing 8: An Xcerpt program making use of the module

We need to declare a hook in the Xcerpt program where the content of a module should be placed. Additionally, we need to provide a complex composition operator that implements the merging of modules. In particular, this operator provides support for configuration of modules.⁵ So that the operator can be invoked, we need to provide a corresponding construct in the extended language. Here we choose to call this construct `import`. Lines 1–3 of Listing 8 show the import statement that Xcerpt has been extended with. Note that we have not explicitly declared the above-mentioned hook. Composition operators may be executed *in-line*. That is, the import statement here represents both the hook and the call to the defined composition operator. The composition operator in this example takes two arguments: the location of the module (`/subClassOf.mxcerpt`) as well as the identifier–fragment pair for the configuration slot in the module (`«rootNode»` to be replaced by `owl`). The steps performed by the import composition operator are as follows.⁶ First, it loads the fragments from the specified module (each of type `XcerptStatement`). Second, it executes the bind composition operator for every slot–fragment pair that the module is parameterized with. Third, it extends the hook represented by the import statement with the loaded fragments. The composition result will thus consist of the two rules

⁵It could also provide support for module encapsulation, however, we will discuss these issues in more detail in a follow-up paper.

⁶The formal definition of the composition operator is not provided here for space reasons.



from the module put together with the goal rule from the program, such that they can be executed by the Xcerpt engine. For space reasons, we do not present the composition result, but it can be found on the Reuseware web page⁷.

The module system we presented here is in a sense rather simple and leaves many aspects of a module system unconsidered. One such aspect is module encapsulation, i.e. making sure that rules from different modules do not affect each other in unintended ways when merged. A composition operator for a module system should perform additional steps to protect, if required and specified, the merged rules from each other. A composition operator extended with just this feature has been implemented for Xcerpt, but is left out from this paper due to space limitations. This extended operator, together with examples, can, however, be found on the Reuseware web page.

4 COMPOSITION FRAMEWORK IMPLEMENTATION

This section describes the *Reuseware Composition Framework* that implements the concepts described above. We use the presented Xcerpt module example to show selected parts of the implemented tool. As one front-end, we implemented a set of plugins for the Eclipse Platform [28]. However, the framework itself can be used independently of Eclipse.

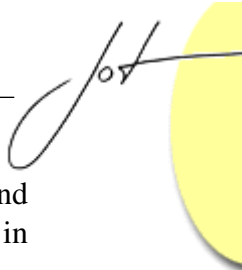
The framework consists of two distinct components. The first component provides tooling to define and extend languages. The second component supports definition of fragments and composition execution.

In the following, we will use the first component to define and extend the Xcerpt grammar. The results of this process (i.e., the generated code) will then be used by the second component to execute the Xcerpt import example.

Generating a Composition System

To get started, we require a description of the original language—in our case Xcerpt. In this paper we use context-free grammars to define languages. Tool support for languages based on context-free grammars does not extend far beyond parser generation. On the other hand, metamodel-based languages have much richer tool support. Therefore, we use EMOF [21] as our implementation technology. The EMOF implementation Ecore and the associated Eclipse Modeling Framework [8] offer a rich and stable tool set to manipulate models and metamodels. Ecore models and metamodels are used to represent fragments and grammars, respectively. Any structure that can be defined by a context-free grammar can also be defined by a metamodel (since context-free grammars describe tree structures and metamodels can also describe graph structures). A mapping from a context-free grammar to a metamodel is possible [1] and is provided by the framework to allow language specification in grammar form. Such a metamodel only describes the syntax of

⁷<http://www.reuseware.org/modularxcerptexample>



a language. The mapping does not take conceptual relationships between grammars and metamodels, as discussed in [16], into account. Such considerations will be included in future work.

A grammatical language description in the tool has two parts. On the one hand, a concrete syntax definition is required in order to process concrete code fragments. On the other hand, the abstract syntax has to be provided, since it clearly defines the grammatical types. Abstract and concrete syntax are cleanly separated into separate files. The abstract description is mapped to an Ecore metamodel from which a Java code representation is generated by the Eclipse Modeling Framework. From the concrete syntax description a parser and a printer are generated utilizing the ANTLR tool set [23]. The separation of abstract and concrete syntax also allows to define several concrete syntaxes, which some languages (e.g. Xcerpt) have, for the same abstract syntax.

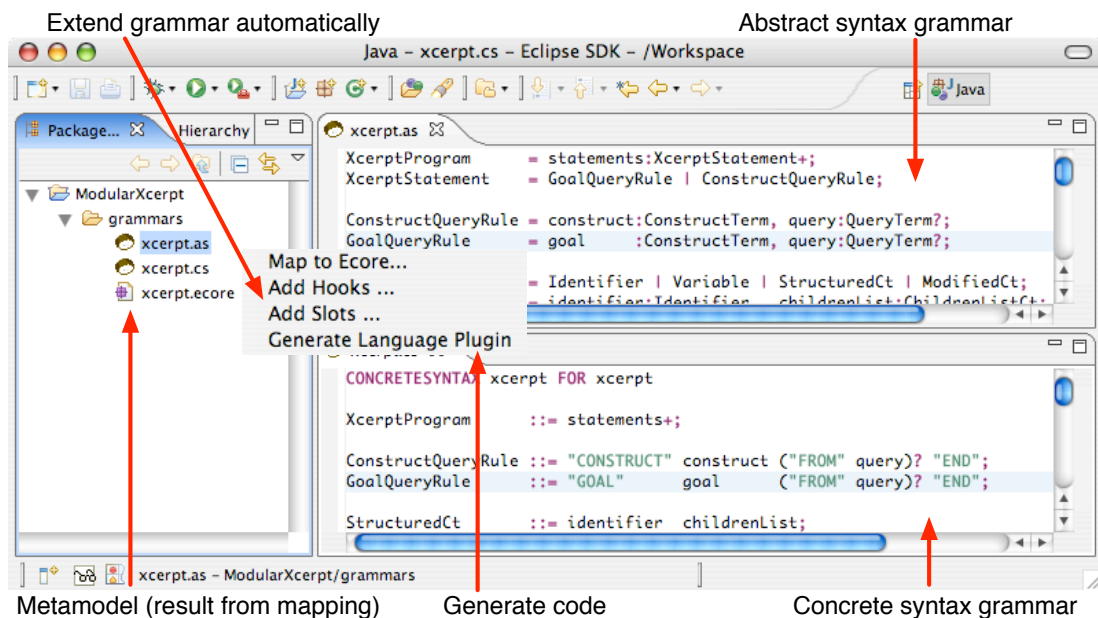


Figure 1: Grammar of Xcerpt in Reuseware

The abstract and concrete syntax grammars of Xcerpt can be derived by separating concrete from abstract elements in the grammar from Listing 2. Additionally, each reference to a production rule in the abstract syntax grammar is tagged with a *role name* (rolename : ProductionRule). These names are then used to annotate the concrete syntax on top of the abstract structure. The production rule for GoalQueryRule (Line 5 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";
```

The Xcerpt grammars as they are used in the tool are shown in Figure 1. After the grammars of the original language have been defined in the tool, they can be extended for reuse. This extension can be done automatically or by manually adding production rules. Additional production rules are annotated to enable the tooling to identify variation points as such and address them during composition. In any case, the extended grammar is a context-free reuse grammar and the extension itself conforms to the transformation function τ (defined in Definition 3). Figure 2 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 repeating the rule name followed by a `==>` and a predefined marker. Such markers are “`componentmodel.Slot`” and “`componentmodel.Hook`”.

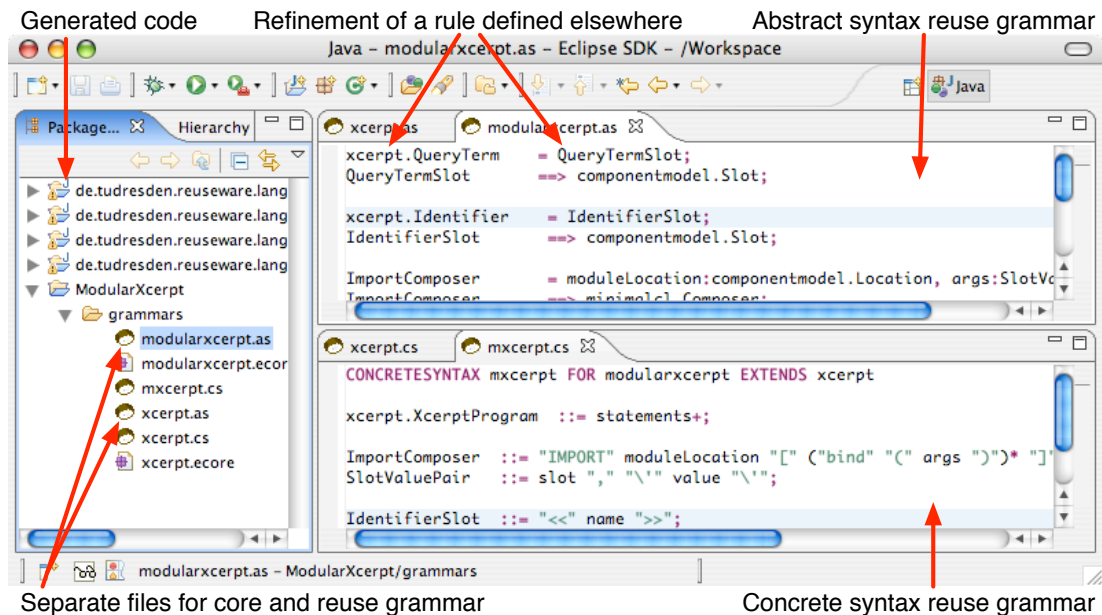


Figure 2: Grammar of ReuseXcerpt in Reuseware

Additional production rules may be placed in a separate file. The completed extended grammar will then consist 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 care about which are original and which are additional rules, since the parsers handle all rules alike.

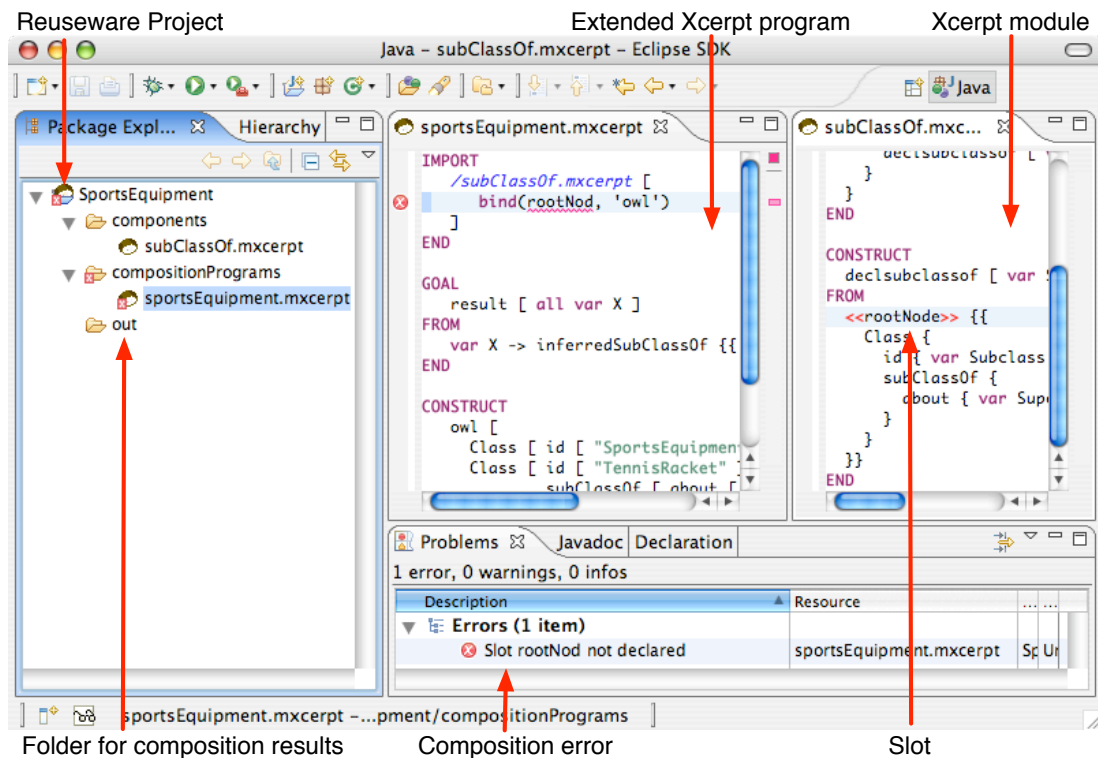


Figure 3: Xcerpt composition in Reuseware

Executing a Composition

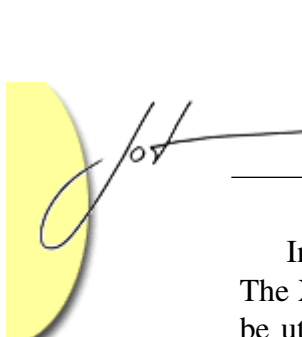
Once plugins are generated, they can be deployed and activated inside the platform. In our case, fragments written in the languages Xcerpt and its reuse extension are now understood by the tooling. Variation points inside underspecified fragments can be recognized and addressed for composition. The composition is executed by merging the abstract syntax trees and using a generated printer to obtain a concrete program as result. Since grammatical types are reflected into the metamodel and into the generated code, type safe composition is ensured. In Figure 3 we show the definition of fragments in the tooling. A *Reuseware Project* provides different folders with distinct behaviors:

components Fragments (possibly underspecified) placed in this folder will be found by the tooling. We place our Xcerpt module here.

composition programs Here, one can place fragments or composition programs⁸. The tooling will automatically interpret the content of this folder and call the corresponding composition operators. We place our extended Xcerpt program here. The tooling will then execute the `import` composition operator.

out After a composition was executed successfully, the result appears here.

⁸A definition of which composition operators are to be applied on which variation points.



In this section we gave a brief overview of the Reuseware Composition Framework. The Xcerpt modular system motivated earlier was realized using the framework and can be utilized as a preprocessor, composing Xcerpt fragments before passing them to the Xcerpt interpreter. Certain details, for instance about composition operator definition, have been left out since they do not directly relate to the theory of composition interfaces presented in this paper. Future work will discuss aspects related to these elements of the framework.

5 RELATED WORK

The Mjølnir System and the Beta language [17] 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, in particular hooks, which can be extended 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 automatically extended with a composition system.

The Software COMPOSITION SysTem (COMPOST) [27] is a predecessor of our current system, which introduced many of the concepts available in our approach, but was limited to Java and XML. Each new language that should be supported by COMPOST, requires a large amount of implementation work. The contribution of this paper is an approach that automates the generation of composition systems for any language that can be specified using a context-free grammar.

Our notion of fragment components is comparable to the notion of *syntactic units* presented in [18]. 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.



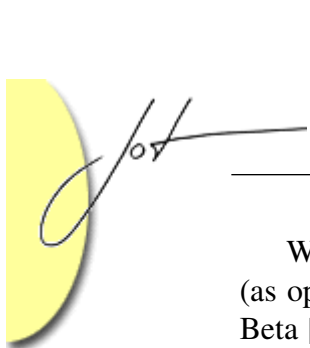
6 CONCLUSIONS AND OUTLOOK

In this paper we have presented a language-independent technology for modularization, composition and reuse. The technology is based on Invasive Software Composition (ISC) [2], a grey-box approach to composition. So far, ISC has been manually implemented for Java and XML [27]. The contribution of this paper is a generic extension of ISC to arbitrary languages. To this end, in Section 3 we have formally defined the relevant concepts of the framework. In Section 4 we have, then, presented an Eclipse-based tool that can automatically generate composition support from language descriptions of arbitrary languages. We have used examples in Xcerpt [7] to demonstrate application of this generic framework to a concrete language.

In this paper, we have only lightly touched on the issue of complex composition operators and composition languages. Additional work has been done in this area, but due to space limitations, this work will have to be reported in a future publication. Further, the approach presented currently supports composition at a syntactic level only. That is, while type-safe composition can guarantee that the composition results will be syntactically valid programs, they may still be semantically invalid. For example, if a programming language requires that a variable be declared before it is used, this cannot be expressed in our typing rules. We are working on enhancing the composition interface of fragments with various types of semantic constraints that will allow to detect such situations through static type-checking of composition programs.

The technology as presented in this paper is based on context-free-grammar descriptions of languages to be extended. However, the underlying concepts—variation points, fragments, and invasive composition—can also be transferred to metamodel-based descriptions of languages. The most important issue one has to deal with here is the fact that metamodel instances tend to have a graph-like structure, while instances of context-free grammars are always trees. The current version of the tool already contains initial support for metamodel-based languages (not shown in this paper). We are going to investigate this further and are planning to provide full support for metamodel descriptions of languages in the near future.

EBNF can itself be described using EBNF. Therefore, our approach can be applied to construct a composition and modularization technique for EBNF itself. Effectively, this provides an opportunity to bootstrap our approach and provide support for language composition. An example, where language composition may be useful is event-condition-action (ECA) languages, often used for describing behavior in a rule-based manner. Depending on context, different sub-languages for events, conditions, and actions may be preferable. However, the core semantics of handling ECA rules—namely, ‘when an event occurs and the corresponding condition holds, then trigger an action’—is the same independent of the concrete sub-languages used. Hence, it would be interesting to study whether we can modularize ECA languages such that we can reuse the core semantics while being able to exchange the different sub-languages as appropriate. The approach presented in this paper may form the technological backbone of such studies.



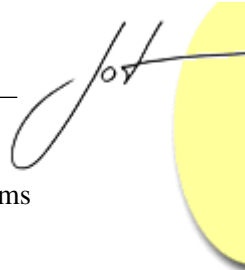
We have presented our work as an approach to static, composition-time composition (as opposed to dynamic run-time composition). However, as can also be seen from the Beta [17] example, there is nothing, conceptually, that stands against dynamic composition of fragments. Dynamic composition of fragments may allow completely new ways of modularizing and dynamically reconfiguring programs. This is still a very open, but very interesting field of research. We hope to be able to cover some ground in this area in the longer-term future.

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 REVERSE number 506779 (cf. <http://reverse.net>), as well as through the 6th Framework Programme project Modelplex contract number 034081 (cf. <http://www.modelplex.org>).

References

- [1] M. Alanen and I. Porres. A relation between context-free grammars and meta object facility metamodels. Technical Report 606, TUCS - Turku Centre for Computer Science, Turku, Finland, Mar 2004.
- [2] U. Abmann. *Invasive Software Composition*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.
- [3] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.
- [4] S. Boag, D. Chamberlin, , M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language. <http://www.w3.org/TR/xquery>, Jan. 2007.
- [5] D. Brickley and R. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation, 10 February 2004. Available at <http://www.w3.org/TR/rdf-schema/>. Accessed 18 November 2006.
- [6] F. Bry, T. Fuche, and S. Schaffert. Initial draft of a language syntax. Technical Report IST506779/Munich/I4-D6/D/PU/a1, Institute for Informatics, University of Munich, 2006.
- [7] F. Bry and S. 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] F. Budinsky, S. A. Brodsky, and E. Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.
- [9] J. Clark. XSL transformations (XSLT). <http://www.w3.org/TR/xslt>, Nov. 1999.



- [10] W. Harrison, H. Ossher, and P. Tarr. Asymmetrically vs. symmetrically organized paradigms for software composition. Technical report, IBM, 2002.
- [11] I. Horrocks and P. F. Patel-Schneider. A proposal for an OWL rules language. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 723–731. ACM, 2004.
- [12] International Organization for Standardization. *ISO/IEC 14977:1996: Information technology — Syntactic metalanguage — Extended BNF*. International Organization for Standardization, Geneva, Switzerland, 1996.
- [13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, Heidelberg, 1997.
- [14] P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering Methodology*, 14(3):331–380, 2005.
- [15] D. C. Kozen. *Automata and Computability*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.
- [16] I. Kurtev, J. Bézivin, and M. Aksit. Technological spaces: An initial appraisal. In *CoopIS, DOA'2002 Federated Conferences, Industrial track*, Irvine, 2002.
- [17] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, June 1993. Reprinted by Mjølner Informatics with permission from Addison-Wesley. Free download at <http://www.daimi.au.dk/beta/Books/betabook.pdf>.
- [18] M. Majkut and B. 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*, Oct. 2003.
- [19] O. Nierstrasz and T. D. Meijler. Research directions in software composition. *ACM Computing Surveys*, 27(2):262–264, June 1995.
- [20] Object Management Group. Unified Modeling Language: Superstructure version 2.0. OMG Document, Aug. 2005. Available at <http://www.omg.org/docs/formal/05-07-04.pdf>.
- [21] Object Management Group. MetaObject Facility (MOF) specification version 2.0. <http://www.omg.org/cgi-bin/doc?formal/2006-01-01>, Jan. 2006.
- [22] H. Ossher and P. 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.
- [23] T. Parr. ANTLR — ANother Tool for Language Recognition — parser generator. <http://www.antlr.org>, Jan. 2007.
- [24] P. F. Patel-Schneider, P. Hayes, and I. Horrocks. OWL web ontology language semantics and abstract syntax. W3C Recommendation, 10 February 2004. Available at <http://www.w3.org/TR/owl-semantics/>.

- [25] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Working Draft 4, October 2006. Available at <http://www.w3.org/TR/rdf-sparql-query/>.
- [26] S. Schaffert, F. Bry, and T. Fuche. Simulation unification. Technical Report IST506779/Munich/I4-D5/D/PU/a1, Institute for Informatics, University of Munich, 2005.
- [27] The COMPOST Consortium. The COMPOST system webpage. <http://www.the-compost-system.org>, Jan. 2007.
- [28] The Eclipse Foundation. Eclipse platform technical overview. <http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html>, April 2006.

ABOUT THE AUTHORS



Jakob Henriksson obtained his MSc. from Linköpings universitet, Sweden, in October 2004. Since July 2005 he is working as research assistant in the Software Technology Group at Technische Universität Dresden. His research interests are in the Semantic Web, software composition and hybrid reasoning. E-mail: jakob.henriksson@tu-dresden.de.



Jendrik Johannes obtained his Dipl.-Medieninf. from Technische Universität Dresden, Germany, in November 2006. Since December 2006 he is working as research assistant in the Software Technology Group at Technische Universität Dresden. He developed the first version of the Reuseware Composition Framework. His research interests are in software, language and (meta)model composition. E-mail: jendrik.johannes@mailbox.tu-dresden.de.



Steffen Zschaler obtained his Dipl.-Inf. from Technische Universität Dresden, Germany, in 2002 and his doctorate in 2007. Currently he is working as a research assistant in the Software Technology Group at Technische Universität Dresden. His research focuses on formal models of non-functional properties of component-based software, and on language and model composition. E-mail: Steffen.Zschaler@tu-dresden.de.



Uwe Assmann is professor for software engineering at Technische Universität Dresden. He received his PhD in Computer Science from Universität Karlsruhe in 1995, showing how to generate program optimizers from graph rewrite systems. In the last years, he has developed an invasive software composition system, COMPOST, for grey-box component composition. E-mail: uwe.assmann@tu-dresden.de.