I3-D12

Typing composition of rule-based languages

Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Project number:	IST-2004-506779
Project instrument:	EU FP6 Network of Excellence (NoE)
Project thematic priority:	Priority 2: Information Society Technologies (IST)
Document type:	D (deliverable)
Nature of document:	R (report)
Dissemination level:	PU (public)
Document number:	IST506779/Dresden/I3-D12/D/PU/b1
Responsible editors:	Jakob Henriksson
Reviewers:	Fabian Abel, Charlie Abela
Contributing participants:	Dresden, Linköping, Warsaw
Contributing workpackages:	I3
Contractual date of deliverable:	29 February 2008
Actual submission date:	7 March 2008

Abstract
This deliverable will describe how it is possible to integrate and apply the typing systems developed within I3 to the composition framework developed within the same group. This will show how the already developed typing techniques can help improve finding errors and problems during composition. The deliverable will in particular describe how a specific composition system addressing Xcerpt benefit from such an integration.

Keyword List
software composition, component based development, semantic web, query languages, Xcerpt, Reuseware, typing

Project co-funded by the European Commission and the Swiss Federal Office for Education and Science within the Sixth Framework Programme.

Typing composition of rule-based languages

Uwe Aßmann[1] and Wlodek Drabent[2,3] and Jakob Henriksson[1] and Artur Wilk[3]

[1] Fakultät Informatik, Technische Universität Dresden
Email: {uwe.assmann|jakob.henriksson}@tu-dresden.de

[2] Institute of Computer Science, Polish Academy of Sciences, Warsaw
Email: drabent@ipipan.waw.pl

[3] Linköpings universitet, Department of Computer and Information Science, Linköping
Email: artwi@ida.liu.se

7 March 2008

Abstract
This deliverable will describe how it is possible to integrate and apply the typing systems developed within I3 to the composition framework developed within the same group. This will show how the already developed typing techniques can help improve finding errors and problems during composition. The deliverable will in particular describe how a specific composition system addressing Xcerpt benefit from such an integration.

Keyword List
software composition, component based development, semantic web, query languages, Xcerpt, Reuseware, typing

Contents

1 Introduction and motivation

The I3 working group—*Composition and Typing*—has developed both composition and typing technology for the rule-based languages developed within REWERSE. The focus has mainly been on the XML and RDF query and transformation language Xcerpt. Typing technology has been developed, enabling static type-checking of Xcerpt programs (see e.g. [3, 13, 14, 15]). The developed typing technology can be used to find errors in programs, which is a great help for query programmers. At the same time, composition technology has been made available that is able to provide languages, such as Xcerpt, with component-oriented language extensions. That is, equip Xcerpt programmers with the possibility to define reusable query entities. For Xcerpt in particular, the module concept was developed, enabling the separation of query programs into reusable and better understood parts (modules) (see [1, 2, 6, 7]). Modules are composed statically, enabling the Xcerpt interpreter to be reused as a black-box when executing modular Xcerpt programs.

The goal of this deliverable is to describe how the techniques developed within the I3 WG can be used together to provide added value for Xcerpt programmers and developers. In particular, we investigate and describe how the composition technology can take advantage of the Xcerpt typing technology when composing Xcerpt programs (and modules). Since both the typing and the composition techniques are static, that is, they operate on program specifications and not during run-time, they are possible to combine. This deliverable does not provide complete integrated tooling or complex examples and use-cases, but describes conceptually how the mentioned added value can be realized.

The structure of this deliverable is as follows. In Section 2 we briefly introduce Xcerpt. In Section 3 we show how types can be useful for Xcerpt. In Section 4 we introduce the module extension to Xcerpt: *Modular Xcerpt*. After having introduced and exemplified the basic parts we then, in Section 5, discuss how they can be integrated to achieve improved robustness for Xcerpt composition. Finally in Section 6 we end the report with some concluding remarks.

2 Xcerpt: A short introduction

There are many publications on Xcerpt (see, e.g., [4, 5, 9]). Here we briefly recall the basics, and instead of attempting an exhaustive description we try to work and explain with intuitive examples. Xcerpt is rule-based and is a language closely related to the logic-programming paradigm. 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 take the form: `CONSTRUCT head FROM body END`. Goal rules make up the output of programs and look 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 *queries* 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.

```
GOAL
        authors [ all author [ var X ] ]
FROM
        book [[ author [ var X ] ]]
END

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

```
 9 END

11 CONSTRUCT
        book [ title [ "Stanley" ], author [ "Tim Jeal" ] ]
13 END
```

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 relating to books is shown in Listing 1. The second and third rules are facts and define two books, each with a title and an author. The first rule defines the output of the program. It queries authors of books, and constructs a list of all found authors.

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. While XML uses "tags", Xcerpt data terms use a square bracket notation. For example, 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. The program in Listing 1 would result in the output: `authors [ author [ "William Dalrymple" ], author [ "Tim Jeal" ] ]`.

Xcerpt queries connect Xcerpt *query terms* via logical connectives (e.g. `or` or `and`). 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.[1] 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 (e.g. `[[ ... ]]` or `{{ ... }}`). 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.

3 Types for Xcerpt

As mentioned in the introduction there has been several publications on types for XML query languages, and Xcerpt in particular (see e.g. [3, 13, 14, 15]). The following section gives an intuitive idea of the benefit of using typing for querying. The running example, based on an example presented in [14], also shows interaction with the developed typechecking prototype system *XcerptT*.[2] The example concerns an imagined online music store. Consider the following Xcerpt program:

[1] This technique is called *simulation unification*, please consult [10] for details.

[2] Can be found online at: `http://www.ida.liu.se/~artwi/XcerptT`.

```
CONSTRUCT
  result [
    all entry [
      var ARTIST,
      all var TITLE
    ]
  ]
FROM
  in { resource [ "file:cds.xml" ],
    bib {{
      cd [[ var TITLE, var ARTIST, "Pop" ]]
    }}
  }
END
```

The program queries the document *cds.xml*. It is intended to extract titles and artists of the CDs of the pop category. Then it should produce a result with entries containing an artist and a list of all his/her CD titles. We assume that a type of the document *cds.xml* and a type of intended query results are given by the following type specification:

```
Cds -> bib[ Cd* ]
Cd -> cd[ Title Artist+ Category? ]
Title -> title[ Text ]
Artist -> artist[ Text ]
Category -> pop | rock | classic

Result -> result[ Entry* ]
Entry -> entry[ Artist Title+ ]

Input::
    resource=file:cds.xml
    typename=Cds

Output::
    typename=Result
```

As a result of typing the program we obtain the following printout:

```
================================================
Type checking ... FAILED: empty result type
------------------------------------------------
Result type: 0
------------------------------------------------
------------------------------------------------
0
================================================
Type Definition:
```

```
-------------------------------------------
Cds -> bib[ Cd* ]
Cd -> cd[ Title Artist+ Category? ]
Title -> title[ Text ]
Artist -> artist[ Text ]
Category -> pop | rock | classic
Result -> result[ Entry* ]
Entry -> entry[ Artist Title+ ]
===========================================
```

The printout says that the typing of the program has failed. Formally, the program is not incorrect w.r.t. the result type specification (as the obtained result type is a subset of the specified type). However the obtained result type is empty. This indicates that the program will not give any results when applied to a document of type *Cd*. This clearly suggests an error. The error is a typo in the query term `cd [[...]]` as *Pop* is written with a capital letter while the type specification requires *pop*.

We correct the program changing *Pop* into *pop* and run the typechecker again. The result is:

```
===========================================
Type checking ... FAILED
-------------------------------------------
Result type: result (not a subset of Result)
-------------------------------------------
Variable-type mappings:
-------------------------------------------
TITLE->Artist, ARTIST->Artist
TITLE->Title, ARTIST->Artist
===========================================
Type Definition:
-------------------------------------------
result -> result[ entry+ ]
entry -> entry[ Artist (Artist|Title)+ ]
Cds -> bib[ Cd* ]
Cd -> cd[ Title Artist+ Category? ]
Title -> title[ Text ]
Artist -> artist[ Text ]
Category -> pop | rock | classic
Result -> result[ Entry* ]
Entry -> entry[ Artist Title+ ]
===========================================
```

This time the inferred result type is not empty but type checking has failed again. This is because the inferred result type *result* is not a subset of the specified result type *Result*, due to type entry not being a subset of type *Entry*. To build data terms of type *entry* the program uses the variable *TITLE*; its values can be of type *Title* or *Artist* (see the "Variable-type mappings" section of the typechecker results). However, in order for the instances of *entry [all ARTIST, all var TITLE]* to be of type *Entry*, the variable *TITLE* has to be bound to data terms of

type *Title*. To correct the query rule we restrict the variable *TITLE* in the rule body so that it is only bound to data terms with the main symbol title. (Thus it can be bound to any element of type *Title*, and to no element of type *Artist*). This is reflected by the following program:

```
CONSTRUCT
  result [
    all entry [
      var ARTIST,
      all var TITLE
    ]
  ]
FROM
  in { resource [ "file:cds.xml" ],
    bib {{
      cd [[ var TITLE -> title{{}}, var ARTIST, "pop" ]]
    }}
  }
END
```

Typechecking of this program shows that it is correct wrt. the type specification:

```
=============================================
Type checking ... OK
---------------------------------------------
Result type: result
---------------------------------------------
Variable-type mappings:
---------------------------------------------
TITLE->Title, ARTIST->Artist
=============================================
Type Definition:
---------------------------------------------
result -> result[ entry+ ]
Cds -> bib[ Cd* ]
Cd -> cd[ Title Artist+ Category? ]
Title -> title[ Text ]
Artist -> artist[ Text ]
Category -> pop | rock | classic
Result -> result[ entry* ]
entry -> entry[ Artist Title+ ]
=============================================
```

The above has showed how type information and type checking can be beneficial to a query language such as Xcerpt.

4 Modular Xcerpt

Component-based development is not only important for traditional programming languages, but also for the languages used on the Web and on the Semantic Web. To enable component-

based development for languages lacking such capabilities the composition framework Reuseware was developed.[3] One of the composition demonstrators was the extension of Xcerpt with the module concept. This extension is referred to as "Modular Xcerpt". Programming in Xcerpt with modules enables separation and reuse of several important querying and transformational aspects, such as:

1. Information extraction (e.g. querying XML-based databases or web-pages).
2. Data transformation (e.g. converting between different serialization formats of RDF).
3. Information presentation (e.g. outputting the queried and transformed data as a web-page customized for a mobile platform, or for a regular desktop computer etc.)

Reuseware implements a composition system for composing Modular Xcerpt programs down into plain Xcerpt. This allows for reusing any engines/interpreters developed for Xcerpt. It should be noted that any properties that are associated with developed modules are guaranteed to be maintained in the composed result. An example of such a property is encapsulation, that is, it is possible to keep the content of modules completely separated, unless it is explicitly expressed that this should not be the case (exploited for example for module communication, that is, for their *interfaces*). For details please see [1].

Here we give an intuitive example of use of Xcerpt modules. Then, in Section 5, we shall show how the typing techniques discussed and demonstrated in Section 3 can be used to improve the robustness of the final result when composing Modular Xcerpt programs. In Section 5 we will explain what kind of errors cannot currently be caught during composition, but which could be found using typing techniques.

4.1 Example – Light-weight ontology reasoning

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 taxonomy (class hierarchy). Ontology reasoners are employed to infer implicit information contained in such ontologies, e.g. to compute the transitive closure of the *subclass-of* relationships. The most prominent ontology language today is the Web Ontology Language OWL [8]. Listing 2 shows an ontology in a simplified OWL syntax modeling sports equipment. The ontology declares that the concept `TennisRacket` is a subclass of the concept `SportsEquipment` and that the concept `WilsonTennisRacket` in turn is a subclass of the concept `TennisRacket`. It should be clear that due to these two facts, the concept `WilsonTennisRacket` is a subclass of the concept `SportsEquipment`, due to the transitivity of the *subclass-of* relationship. However, this information is only implicitly contained in the ontology and an ontology reasoner is needed to infer this knowledge. However, as we shall see, an Xcerpt program can also be used to infer such simple implicit ontology information.

```
<?xml version="1.0" ?>
<owl>
       <class id="SportsEquipment" />
       <class id="TennisRacket">
              <subclassof about="SportsEquipment" />
       </class>
       <class id="WilsonTennisRacket">
              <subclassof about="TennisRacket" />
```

[3] http://www.reuseware.org

```
      </class>
</owl>
```

Listing 2: An example ontology located in file `file:sports.owl` modeling sports equipment in a simplified OWL syntax.

The two rules in Listing 3 shows an Xcerpt program that queries an OWL ontology (in this case the `sports.owl` document shown in Listing 2) and calculates all the implicit subclass-of relationships contained in the ontology.

```
GOAL
  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
  in { resource { "file:sports.owl", "xml" },
    owl {{
      class {{ attributes { id { var Subclass } },
        subclassof {{ attributes { about { var Superclass } } }}
      }}
    }}
  }
END
```

Listing 3: An Xcerpt program querying an OWL ontology and deriving implicit *subclass-of* relationships.

The result of executing the program in Listing 3 is shown in Listing 4. It shows that the implicit *subclass-of* relationship between concepts *WilsonTennisRacket* and *SportsEquipment* was derived.

```
inferredSubClassOf [
  subClassOf [
    "TennisRacket",
    "SportsEquipment"
  ],
  subClassOf [
    "WilsonTennisRacket",
    "TennisRacket"
  ],
  subClassOf [
    "WilsonTennisRacket",
    "SportsEquipment"
  ]
]
```

Listing 4: Result after executing Xcerpt program shown in Listing 3.

The program in Listing 3 is surely desirable to be reused across different Xcerpt programs whenever such simple ontology reason is desired. That is, it would be desirable to make it a

reusable Xcerpt module. This is possible in the Xcerpt extension Modular Xcerpt. The rules in Listing 5 describe such a reusable Xcerpt module, adapted and generalized from the program in Listing 3. As can be seen, an Xcerpt module is essentially a set of related Xcerpt construct rules with explicitly defined interfaces for communication with modules or programs importing the module.

We recognize that the Xcerpt rules in Listing 5 is a module because of the used keyword `MODULE`. A module is given a name, which here is `subClassOf`. The module shown here is generalized from the one in Listing 3 in the sense that a specific ontology is no longer queried, but the module instead assumes some input "format", which can be seen in the query part of the second rule. Each module is enforced to be separate from every other module, hence, each module is encapsulated. This means that no two rules in different modules depend on each other. The new keyword `public` is used to break the default encapsulation for the purpose of module interfaces. The keyword may precede a top-level query or construct term. Using the `public` keyword before a query expresses that data is required for the module to perform its service, that it expects data to operate on. A rule with a public query can be seen as an "input rule". Using the `public` keyword before a construct term expresses that the module provides some output that may be processed by a module or program importing the module in question. Hence, we notice that the module in Listing 5 both requires and provides data. The module requires a list of explicit *subclass-of* relationships and provides in addition the implicit *subclass-of* relationships.

```
MODULE 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 5: An Xcerpt module in the file `/subclassof.mxcerpt`.

It should be clear that the keywords `MODULE` and `public` extend Xcerpt and are introduced to define modules. There are also new constructs for importing (declaring) and using already defined modules, hence reusing them. An Xcerpt program using the module defined in Listing 5 is shown in Listing 6. The program uses the module to find all super-classes of the concept

WilsonTennisRacket. In the ontology from Listing 2, there is one explicit and one implicit super-class to this concept. The explicit one can be found directly, but the implicit one is found using the reasoning capabilities provided by the imported module.

```
1  IMPORT /subclassof.mxcerpt AS rdfs

3  GOAL
     wilsonsuperclasses [ all var Super ]
5  FROM
     in rdfs (
7      rdfsengine {{
         output {{
9          inferredsubclassof {{
             subclassof [ "WilsonTennisRacket", var Super ]
11         }}
         }}
13     }}
     )
15 END

17 CONSTRUCT
     to rdfs (
19     rdfsengine [
         input [
21         explicitsubclassof [ var Subclass, var Superclass ]
         ]
23     ]
     )
25 FROM
     in { resource { "file:sports.owl", "xml" },
27     owl {{
         class {{ attributes { id { var Subclass } },
29         subclassof {{ attributes { about { var Superclass } } }}
         }}
31     }}
     }
33 END
```

Listing 6: An Xcerpt program making use of a module.

Three constructs are introduced for the purpose of importing and using modules. First, the `IMPORT` construct for actually importing reusable modules. Then, a *to-module* and an *in-module* construct for communicating with imported modules. The program in Listing 6 hence, on Line 1, imports the module from Listing 5. On Line 18 the program provides data to the imported module, and on Line 6 the data transformed by the module is again queried.

Notice that in Listing 6, which OWL document is being queried, and the way it is queried is directly encoded in the user program, and not in the module. First of all, this makes the module more general since it is not directly bound to OWL *subclass-of* relationships. Furthermore, we can make the example even more flexible by encapsulating how OWL in particular is queried in yet another module, but this is not demonstrated here. The above provides an example and conveys the idea of how modules can be used in Xcerpt. For more detail, please see [1].

5 Typing compositions

The basic motivation for investigating the role types can play in the composition framework Reuseware[4] is due to an underlying restriction in the framework. The composition framework is based on Invasive Software Composition (ISC), which is a static, fragment-based, composition

[4] `http://www.reuseware.org`

approach [11]. That is, entities being composed are source-code fragments of some underlying *base language*. The framework provides one important guarantee on the composition result, namely, that it is a valid sentence of the base language. This means that the composition result will always be syntactically correct wrt. the base language. However, such guarantees are often not enough. Syntactically valid programs often contain semantic errors which are not intended by the programmer. In a similar way it is possible to use Reuseware to compose programs that are valid wrt. the base language syntax, but that are semantically invalid (either wrt. the base language or the intention of the programmer).

In Section 3 we saw how typing techniques can be used to find the errors in such semantically ill-defined programs. In the same way, we want to employ such typing techniques to find ill-defined compositions. This would hence improve the robustness of compositions by not only guaranteeing syntactically correct composition results, but also covering cases that go beyond: semantically correct compositions.

5.1 Module Contracts

An Xcerpt module can be said to encapsulate some functionality or service. To perform this service the module may require some input (data terms), and always provides some output (data terms). To improve robustness of module use (how input is provided to the module and how output produced by the module is used in a module-importing program), it is desirable to associate a *contract* with the module, provided by the module developer and describing how the module is to be successfully used. The formalisms chosen to specify such contracts is Xcerpt types. A module with a contract can look like the module in Listing 7 (comments after #).

```
MODULE moduleName

CONTRACT
(
        Input: ModuleInputType
        Output: ModuleOutputType
        Types: file:type.xts
)

# "module output interface" rule
CONSTRUCT
        public outputTerm [ ... ]  # annotated to provide "outgoing" data terms
FROM
        # query ...
END

<more Xcerpt rules>

# "module input interface" rule
CONSTRUCT
        # construct term ...
FROM
        public inputTerm [ ... ]  # annotated to accept "incoming" data terms
END
```

Listing 7: An Xcerpt module with an associated contract.

A module contract, like the one showed in Listing 7, specifies:

1. A type specification (in XcerptT internal type specification format)
2. A type for the required input data terms (from the associated type specification)

3. A type for the provided output data terms (from the associated type specification)

The above program contains two "interface rules", one for the module output (construct term annotated with keyword `public`) and one for module input (query term annotated with keyword `public`). An interface rule specifies that this rule will be used in data sharing with programs (or other modules) importing this module. In the general case a module can contain more than one input and one output interface rule. However, in the remaining parts we consider the specific case of a module having only one input and one output interface rule. Thus, each module has one (and only one) rule with a public annotated construct term and one (and only one) rule with a public annotated query term.

5.2 Verifying module definition correctness

A module is checked wrt. its contract when being developed, before it is being used (that is, imported). As has been mentioned, a module can in many respects be seen as a normal Xcerpt program, that is, a set of Xcerpt rules. However, with the important difference that some rules are explicitly stated to be module input or output interface rules (using the `public` keyword), and that a contract is associated with the module. This means that verifying a module wrt. its contract is, as we shall see, very similar to type checking an Xcerpt program as was demonstrated in Section 3.

For convenience we use the notation $c \leftarrow q$ for construct rules and $g \leftarrow q$ for goal rules, where c, g are construct terms and q a query. Let us represent an arbitrary Xcerpt module M by (1) (p for "public" and rl for a rule).

$$\underline{p}\ c_1 \xleftarrow{rl_1} q_1, c_2 \xleftarrow{rl_2} q_2, \ldots, c_n \xleftarrow{rl_n} \underline{p}\ q_n \tag{1}$$

We assume the module has one public query and one public construct term (those preceded by p). The module (1) is verified wrt. its associated contract by transforming it into Xcerpt program M' in (2) (r for external resource with "file:dummy.xml" and "xml" as parameters), where $g_1^{M'} = c_1^M$.

$$g_1 \xleftarrow{rl_1} q_1, c_2 \xleftarrow{rl_2} q_2, \ldots, c_n \xleftarrow{rl_n} in[r, q_n] \tag{2}$$

Since the transformed module M' in (2) is a plain Xcerpt program, we can reuse the Xcerpt type checker for verifying it wrt. the M-associated contract. This is done by sending the following typing information to the type-checker XcerptT:

```
<type spec> (from e.g. file:type.xts)

Input::
     resource=file:dummy.xml
     typename=ModuleInputType

Output::
      typename=ModuleOutputType
```

The type specification referenced from the module contract should be inserted where it says `<type spec>`, and the module input type (here: `ModuleInputType`) and output type (here: `ModuleOutputType`) are also assumed to be the ones from the contract. If the type checker

confirms type correctness, the module is valid wrt. its contract. A module being valid wrt. its contract can be understood as: If module input data is of specified input type, then module output data is of specified output type. Note that if the type checker does not confirm type correctness, the module *may* be invalid wrt. its contract. In this case, the type incorrectness cannot be guaranteed since the types are approximations (see [12] for details). Nonetheless, a non-valid module may be a signal to the module developer that something is wrong.

Example 1 Consider the Xcerpt module in Listing 8. It is the module from Listing 5, but augmented with a contract.

```
MODULE subClassOf

CONTRACT
(
        Input: Rdfsin
        Output: Rdfsout
        Types: file:rdfstype.xts
)

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 8: An Xcerpt module with a contract.

The file `file:rdfstype.xts` contains the following type definition.

```
Rdfsin -> rdfsengine [ Input ]
Input -> input [ Explicit ]
Explicit -> explicitsubclassof [ Text Text ]
Rdfsout -> rdfsengine [ Output ]
Output -> output [ Inferred ]
Inferred -> inferredsubclassof [ Subclass* ]
Subclass -> subclassof [ Text Text ]
```

To check if the module is valid wrt. its contract, we transform the module in Listing 8 to the module in Listing 9.

```
GOAL
  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
  in { resource [ "file:dummy.xml", "xml" ],
    rdfsengine [
      input [
        explicitsubclassof [ var Subclass, var Superclass ]
    ] ]
  }
END
```

Listing 9: A plain Xcerpt program.

The transformed (plain) Xcerpt program in Listing 9 is then checked wrt. its contract, using the type information below.

```
Rdfsin -> rdfsengine [ Input ]
Input -> input [ Explicit ]
Explicit -> explicitsubclassof [ Text Text ]
Rdfsout -> rdfsengine [ Output ]
Output -> output [ Inferred ]
Inferred -> inferredsubclassof [ Subclass* ]
Subclass -> subclassof [ Text Text ]

Input::
      resource=file:dummy.xml
      typename=Rdfsin

Output::
      typename=Rdfsout
```

As can be seen below, the type checker confirms that the module is valid.

```
Loading type specification rdfstype.xts ...
======================================================================
Rule 1: rdfsengine
Type checking: OK
----------------------------------------------------------------------
Superclass->Text, Subclass->Text
```

```
Z->Text, Subclass->Text, Superclass->Text

======================================================================
Rule 2: declsubclassof
----------------------------------------------------------------------
Superclass->Text, Subclass->Text

======================================================================
======================================================================
Type Definition:
----------------------------------------------------------------------
rdfsengine -> rdfsengine[ output ]
output -> output[ inferredsubclassof ]
inferredsubclassof -> inferredsubclassof[ subclassof+ ]
declsubclassof -> declsubclassof[ Text Text ]
Rdfsin -> rdfsengine[ Input ]
Input -> input[ Explicit ]
Explicit -> explicitsubclassof[ Text Text ]
Rdfsout -> rdfsengine[ Output ]
Output -> output[ Inferred ]
Inferred -> inferredsubclassof[ subclassof* ]
subclassof -> subclassof[ Text Text ]
======================================================================

Evaluation took 0,10s
```

Thus, we say that the module is valid wrt. its contract.

5.3 Verifying module usage correctness

When using modules, not only can we reuse the service they provide, but we can also reuse the contract verifications done during module definition, as explained in Section 5.2. We will here give an example of how this can be done.

Example 2 Consider the program in Listing 6, making use of the module we just verified to be valid wrt. its contract above. Consider the type specification below, consisting of three "parts". The first part is from the used module contract. The second part is the type specification for our simplified OWL format. The last part is the specification for our desired result type.

```
Rdfsin -> rdfsengine [ Input ]
Input -> input [ Explicit ]
Explicit -> explicitsubclassof [ Text Text ]
Rdfsout -> rdfsengine [ Output ]
Output -> output [ Inferred ]
Inferred -> inferredsubclassof [ Subclass* ]
Subclass -> subclassof [ Text Text ]

Owl -> owl [ Class* ]
```

```
Class -> class [ AttrId SubclassOf ]
AttrId -> attribut [ Id ]
Id -> id [ Text ]
SubclassOf -> subclassof [ AttrAbout ]
AttrAbout -> attribut [ About ]
About -> about [ Text ]

WilsonSupClasses -> wilsonsuperclasses [ Text* ]
```

The program in Listing 6 contains one rule using the *to-module* construct. Thus, it is producing input data for the module. That rule can be transformed into the (plain) Xcerpt program in Listing 10.[5]

```
GOAL
  rdfsengine [
    input [
      explicitsubclassof [ var Subclass, var Superclass ]
    ]
  ]
FROM
  in { resource [ "file:sports.owl" ],
    owl {{
      class {{ attribut { id { var Subclass } },
        subclassof {{ attribut { about { var Superclass } } }}
      }}
    }}
  }
END
```

Listing 10: Transformed rule for checking that data sent to the module is of valid type.

The only difference is that we have a goal rule instead of a construct rule and that we have removed the reference to the module. The Xcerpt program in Listing 10 can then be type checked wrt. the type specification above using the following specification for the input and output type:

```
Input::
      resource=file:sports.owl
      typename=Owl

Output::
      typename=Rdfsin
```

The specified input type is the type of the data we are querying, that is, the simplified OWL format. As for the output type we specify the input type of the used module contract. When deploying the type checker we get the following result:

```
======================================================================
Rule 1: rdfsengine
Type checking: OK
----------------------------------------------------------------------
```

[5] We have changed the label `attributes` to `attribut` in this example for some technical reasons due to the type checker.

```
Subclass->Text, Superclass->Text

========================================================================
========================================================================
Type Definition:
------------------------------------------------------------------------
rdfsengine -> rdfsengine[ input ]
input -> input[ explicitsubclassof ]
explicitsubclassof -> explicitsubclassof[ Text Text ]
Rdfsout -> rdfsengine[ Output ]
Output -> output[ Inferred ]
Inferred -> inferredsubclassof[ Subclass* ]
Subclass -> subclassof[ Text Text ]
Owl -> owl[ Class* ]
Class -> class[ AttrId SubclassOf ]
AttrId -> attribut[ Id ]
Id -> id[ Text ]
SubclassOf -> subclassof[ AttrAbout ]
AttrAbout -> attribut[ About ]
About -> about[ Text ]
========================================================================

Evaluation took 0,20s
```

The result says that the rule is type correct. This means: If the OWL document we are querying is of the specified input type, then the data produced for the module is a sub-type of the module's required input type. Thus, we know that it will be possible for the module to use the data we are producing for it.

Likewise, we can check if the data produced by the module is correctly used by the program importing the module. The other rule in Listing 6, making use of the *in-module* construct, queries the module. That rule can be transformed to the one in Listing 11.

```
GOAL
  wilsonsuperclasses [ all var Super ]
FROM
  in { resource [ "file:dummy.xml", "xml" ],
    rdfsengine {{
      output {{
        inferredsubclassof {{
          subclassof [ "WilsonTennisRacket", var Super ]
        }}
      }}
    }}
  }
END
```

Listing 11: Transformed rule for checking that data produced by the module and queried by the program is of expected type.

Instead of actually querying the module with the *in-module* construct we query a dummy resource (here: `file:dummy.xml`). We then associate the output type of the module we are using with the dummy resource, as follows:

```
Input::
      resource=file:dummy.xml
      typename=Rdfsout

Output::
      typename=WilsonSupClasses
```

As output type we of course specify the desired output type. Deploying the type checker on the rule in Listing 11, using the described type specifications, we get the following result:

```
======================================================================
Rule 1: wilsonsuperclasses
Type checking: OK
----------------------------------------------------------------------
Super->Text

======================================================================
======================================================================
Type Definition:
----------------------------------------------------------------------
wilsonsuperclasses -> wilsonsuperclasses[ Text+ ]
Rdfsin -> rdfsengine[ Input ]
Input -> input[ Explicit ]
Explicit -> explicitsubclassof[ Text Text ]
Rdfsout -> rdfsengine[ Output ]
Output -> output[ Inferred ]
Inferred -> inferredsubclassof[ Subclass* ]
Subclass -> subclassof[ Text Text ]
Owl -> owl[ Class* ]
Class -> class[ AttrId SubclassOf ]
AttrId -> attribut[ Id ]
Id -> id[ Text ]
SubclassOf -> subclassof[ AttrAbout ]
AttrAbout -> attribut[ About ]
About -> about[ Text ]
WilsonSupClasses -> wilsonsuperclasses[ Text* ]
======================================================================

Evaluation took 0,10s
```

The type checker shows that the program is valid wrt. is type specification. This means: If the module produces data of the specified type, then the data produced by the program will be of the specified type.

Notice that we have checked the correctness of Listing 6 independently of the rules in the module. That is, we have used the assumption that the module is type correct wrt. its contract when checking the program that uses the module.

We can now for example, statically, during module composition, find errors in how modules are used. Consider for example misspelling a label in the rule that produces data for the module. Then we would construct the rule in Listing 12 to be type checked.

```
GOAL
  rdfsengine2 [
    input [
      explicitsubclassof [ var Subclass, var Superclass ]
    ]
  ]
FROM
  in { resource [ "file:sports.owl" ],
    owl {{
      class {{ attribut { id { var Subclass } },
        subclassof {{ attribut { about { var Superclass } } }}
      }}
    }}
  }
END
```

Listing 12: Transformed rule for checking that data sent to the module is of valid type.

We would then receive the following result from the type checker:

```
======================================================================
Rule 1: rdfsengine2
Type checking: Failed (no results of type Rdfsin)
----------------------------------------------------------------------
Subclass->Text, Superclass->Text

======================================================================
======================================================================
Type Definition:
----------------------------------------------------------------------
rdfsengine2 -> rdfsengine2[ input ]
Rdfsin -> rdfsengine[ input ]
input -> input[ explicitsubclassof ]
explicitsubclassof -> explicitsubclassof[ Text Text ]
Rdfsout -> rdfsengine[ Output ]
Output -> output[ Inferred ]
Inferred -> inferredsubclassof[ Subclass* ]
Subclass -> subclassof[ Text Text ]
Owl -> owl[ Class* ]
Class -> class[ AttrId SubclassOf ]
AttrId -> attribut[ Id ]
Id -> id[ Text ]
SubclassOf -> subclassof[ AttrAbout ]
AttrAbout -> attribut[ About ]
About -> about[ Text ]
======================================================================

Evaluation took 0,30s
```

This result can be used to report to the module programmer (at composition-time) that there is a potential error in how the module is used.

6 Conclusions

We have in this deliverable described how it is possible to take advantage of type information and type checking during composition of Xcerpt programs. Being able to do this helps in finding errors in programs at composition-time, that is, statically. In the same way that type information in general can be useful to programmers in a normal programming setting, types can also help a composition environment to provide better feed-back to users in how components (here: modules) are used.

References

[1] U. Aßmann, S. Berger, F. Bry, T. Furche, J. Henriksson, and J. 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.

[2] U. Aßmann, S. Berger, F. Bry, T. Furche, J. Henriksson, and P.-L. Patranjan. A generic module system for web rule languages: Divide and rule. In *Proc. Int'l. RuleML Symp. on Rule Interchange and Applications*, 2007.

[3] S. Berger, E. Coquery, W. Drabent, and A. Wilk. Descriptive typing rules for xcerpt. In *PPSWR*, pages 85–100, 2005.

[4] F. Bry and S. Schaffert. A gentle introduction into xcerpt, a rule-based query and transformation language for xml. In *Proceedings of International Workshop on Rule Markup Languages for Business Rules on the Semantic Web, Sardinia, Italy (14th June 2002)*, 2002.

[5] Francois 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, London, UK, 2003. Springer-Verlag.

[6] J. Henriksson, F. Heidenreich, J. Johannes, S. Zschaler, and U. Aßmann. Extending grammars and metamodels for reuse – the reuseware approach. *To Appear in IET Software, Special Issue on Language Engineering*, 2007.

[7] J. Henriksson, J. Johannes, S. Zschaler, and U. 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.

[8] 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/`.

[9] S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web.* Dissertation/Ph.D. thesis, Institute of Computer Science, LMU, Munich, 2004. PhD Thesis, Institute for Informatics, University of Munich, 2004.

[10] 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.

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

[12] A. Wilk. *Types for XML with Application to Xcerpt*. PhD thesis, Linköping University, Department of Computer and Information Science, 2008.

[13] A. Wilk and W. Drabent. On types for xml query language xcerpt. In *PPSWR'03*, pages 128–145, 2003.

[14] A. Wilk and W. Drabent. A prototype of a descriptive type system for xcerpt. In *PPSWR*, pages 262–275, 2006.

[15] Wlodek Drabent and Artur Wilk. Type inference and rule dependencies in Xcerpt. 2007. Submitted to RR'07.