Rewriting For Parametrization

Markus Lepper[1] and Baltasar Trancón y Widemann[2]

[1] <semantics/> GmbH, Berlin
post@markuslepper.eu
[2] Ilmenau University of Technology, DE
Baltasar.Trancon@tu-ilmenau.de

Abstract. In most computer languages, parametrization of library modules is realized by pre-wired parameters, which must be instantiated to some concrete value when importing. This approach is not optimal for text structure definitions (aka "document types"), used in document processing. The situation is special since (a) all definitions contained are totally visible to the user anyhow ("glass-box"), and (b) potentially every reference can come into the need of being a parameter, when application contexts evolve. Therefore a method of free rewriting seems more adequate. This article presents the outlines of the module rewriting system as implemented in the authors' d2d language for notating XML encoded documents. The theoretical and technical problems for pure algebraic usage (here: control of a parsing process) are trivial, but not for reifying and referring to the rewriting results in a sensible way, which is necessary for generating diagnosis and user documentation. The article gives an algorithm for both cases and discusses the transfer to other languages with glass-box approach.

Keywords: Rewriting; Modularization; Text Processing; Document type

1 Introduction

The d2d project realizes a front-end for directly writing down XML-encoded documents, in the flow of creative authoring. The necessary text structure definition is a collection of content definitions per each single element type, as usual. These definitions are organized in modules, which are imported and parametrized as a whole. In the last years very different applications of d2d evolved, from musicology to book-keeping. Most of their text structure definitions rely on one general purpose base architecture, called `d2d_gp`. Its components and features resemble the LaTeX article style [4], with lists, tables, floats, tables of contents, footnotes, abstracts, hyper-refs, citations, etc. and it as a whole, or parts of it, have been imported, parametrized and adopted to serve those very different purposes.[3]

It turned out that this kind of reuse is hardly feasible with particular components being pre-wired for parametrization, but that the need of modifying

[3] See `http://senzatempo.de/mahler/gmahler_sinf3_satz1.html` and `http://bandm.eu/metatools/docs/usage` for very different instantiations.

or exchanging subdefinitions arises nearly everywhere in the structure. For this very reason, among others, famous systems like TEX [3] and LATEX [4] employ *dynamic binding*, to achieve the necessary flexibility for reuse. But this mechanism is neither explicit nor compositional; the intended interactions can neither be checked automatically, nor easily understood or documented. So the d2d module system is based on a third alternative, on explicit *free rewriting*: Every reference appearing in an expression in any definition can be altered when importing and instantiating the containing module. Astonishingly, this simple idea is novel in this context.

Coupling rewrite rules with module imports allows to adopt a whole collection of definitions in an atomic and consistent way. Basically, this mechanism is easy to understand, straight-forward to implement and sufficiently expressive, as our tool has proven in above-mentioned productive contexts.

The results of the rewriting are sentences in what is called *underlying domain language* in the following. With these, two fundamentally different use cases must be distinguished. In the first, called *algebraic* in the following, the results of rewriting are used to control the execution of some evaluation process (in our case, where the underlying domain language is a grammar description, this is a parsing process). This case is easy to realize, as the discussion will show.

Severe problems arise in the second case, called *reified* in the following. Here the results of rewriting shall be *identified* in a sensible way, to generate statistics, user documentation, interactive error diagnosis, etc. All these are central issues in projects which aim at domain experts not necessarily language experts. Here an ergonomically sensible and computable notion of *equivalence* of the parametrized element definitions is required.

The contributions of this article are practical in two concerns: First, we give operational semantics for both cases, which are immediately executable and designed for optimized performance. The first case is basically straight forward, but there are some tricky details related to practical application. For the second case is we establish a clean separation of a generic part and of the specifics of the underlying domain language.

Second, many design decision have come from concrete needs in daily practice. These will be mentioned in the following, partly in footnotes.

Beyond the original context the results are applicable to other underlying domain languages which follow a *glass-box approach*, i.e. the user who instantiates a module sees the internal structure of the contained definitions. This is of course the case with all kinds of document type definitions, but it applies also to other realms like test data partitioning trees, libraries of grammar rules for natural language processing, collections of algebraic/logic axioms for proof derivation, openMath phrasebooks, text-based workflow languages, constraint pools for real-world problem solving, models in description logics, etc. In all these contexts our approach would be sensible.

2 The d2d Context

2.1 Principles of parsing

The d2d language is a front-end for creating complex documents, encoded as XML objects. Intended for authoring and typing by domain experts, it tries not to disturb the mental flow of writing, by minimalization of input noise. Therfore it is equally well *readable* for humans, and suited for voice input. It combines two layers of input: The top layer employs explicit tagging, the lower realizes totally implicit tagging by character based parsers. On the tag level, most close tags and some open tags are inferred and need not to be typed.

These are typical first lines of a d2d source text:

```
#d2d 2.0  text using  basic.deliverables : webpage
#title Example of a Web Page
#date 2012-12-28
#authors #author Kraus, Karl #author Karl von Valentin
#lang en+de
```

This input will be translated to ...

```
<webpage xmlns:a="http:bandm.eu/d2d_gp/basic/deliverables"
         xmlns:b="http:bandm.eu/d2d_gp/basic/personalNames_de">
  <a:title>Example of a Web Page</title>
  <a:date><a:year>2012</a:year><a:month>12</a:month>
<a:dayOfMonth>28</a:dayOfMonth></a:date>
  <a:authors>
    <a:author><b:name><b:given>Karl</b:given>
                <b:family>Kraus</b:family></b:name></a:author>
    <a:author><b:name><b:given>Karl</b:given><b:con>von</b:con>
                <b:family>Valentin</b:family></b:name></a:author>
  </a:authors>
  <a:langs><a:lang>en</a:lang><a:lang>de</a:lang></a:langs>
```

The genuine XML syntax cannot be offered to authors who are used to the act of writing as an immediate expressions of their personality, in the mental state of creative flow. Also syntax controlled text editors, with all their blinking, popping-up, interfering and colouring, are not an alternative for these truly creative. Especially the character parsers, here for author, date, etc., disburden the writer massively, nevertheless yielding well structured output in a standardized format. This example is only meant to give a fundamental impression; exact specifications can be found in [6] and [5].

2.2 Element Structure Definitions

A definition of the text structure is required for steering the parsing process. The current implementation of d2d does understand W3C DTD [1], but our own text structure libraries are written in d2d's proprietary definition language, heavily employing its two genuine features, the definition of character parsers and module import with free rewritings.

Every module maps identifiers to definitions. (See the data types *Module* and *Definition* in Table 1; the mathematical notation applied is summarized in Appendix A.) Each definition corresponds to an XML element type in the generated output, where its identifier serves as its XML tag. A definition can carry auxiliary XML specific parameters like "attribute" or "element" storage type, tag override, namespace URL, trimming of whitespace, multi-lingual user documentation, XSLT rules for the translation into various back-ends, etc.[4]

But mainly a definition contains a regular expression from T in Table 1 which defines the possible contents of the element and governs its parsing process. Its fundamental constructors, their meanings and constraints are very similar to those in genuine XML DTD [1][5] and RelaxNG [2]. The constructors "`,`", "`|`", "`?`", "`&`", etc. have their usual meanings.

The basic atoms of the expressions are references from R. These are either simple identifiers from $\mathbb{I}_T$, and refer to the definitions contained in the same module as the expression, or identifiers prefixed with one or more import keys from $\mathbb{I}_I$. Then they refer to a definition in an imported module.

2.3 Modules and Parametrization

Beside element definitions, every module can contain import clauses, modeled by the data type *Import* in Table 1. For each of these (1) an import key is defined (from $\mathbb{I}_I$), to be used as a prefix for referring to the definitions in the imported module; (2) its source text is identified by an absolute module address from $\mathbb{I}_A$.[6]; and (3) three different kinds of *rewrite rules* may be applied to this source:

Global rewritings replace every occurrence of a particular reference everywhere in the imported module by the given *replacement expression*. This applies to both kinds of expressions in the imported module, namely in element contents definitions and in replacement expressions in import clauses. The replacement expression will be evaluated in the context of the importing module.

Local rewritings only affect one single element definition in the imported module, as indicated by its identifier. In its expression, the given reference is replaced by the replacement expression in the same way as with a global rewriting. A local rewriting overrides the global rewriting for the same reference.

[4] Most of these additional parameters need not to be known by those users who are mere text authors. The glass-box is not totally transparent. Most important: the XSLT rules which come with most elements for translation into different back-ends are copied "under the hood" when importing modules.

[5] Like DTD, d2d requires "1-unambiguity".

[6] In the concrete implementation, the address of a module $\mathbb{I}_A$ is not a simple identifier but a dot-separated path, corresponding to static nesting of modules. This is one of the details we do abstract from in this article. Similar with $\mathbb{I}_T$: Tags in the input document may have local scope to the containing element. This corresponds to definitions local to definitions, which are referred to also by dot-separated paths.

Import rewritings replace the whole import clause in the imported module with the given import key by an import clause defined in the importing module.

In the d2d format this looks like

```
1  module uses_other_modules
2    import M from module_to_import
3           ^( (M.a, M.J.L.b, c)* / MODULE_PARAMETER )
4           ^( N / H )
5    import N from another_module
6           in a ^( (N.a, N.a) / a )
7           in b ^( a / a )
```

This example module imports a first module and replaces in its source, according to the rewrite rule in line 3, all references to "MODULE_PARAMETER" by the given expression. This in turn contains a reference "M:a" to the imported and parametrized module itself. Please note that "MODULE_PARAMETER" is syntactically the same as a simple identifier "a"; its appearance has been chosen only to indicate to the user that it is *foreseen* to be rewritten: Parametrization in the narrow sense is subsumed as a special case of rewriting.[7]

The second import rewrites the regular expressions of the definitions "a" and "b" in the imported module. In "a" it rewrites every reference to itself to a sequence of two of these (line 6). In "b" it rewrites all references to "a" to a definition named "a" in the importing module (line 7): In "a/a" the second term is the reference to be replaced, as it stands unevaluated in the source representation of the imported module; the first term is the replacement expression, evaluated in the context of the importing module.

In most concrete fields of application, *implicit re-exports* turned out to be indispensable: In line 3 above "M.J.L.b" does refer to a definition in an imported module of an imported module of an imported module; another aspect of the glass-box approach. [8]

The third kind of rewriting is that of a whole import clause. It has been introduced especially for multi-lingual support: All structure definitions e.g. for calendric date and time, or for personal names, or postal addresses, are exchanged together, by one single rewrite rule, switching between whole modules. Currently this feature is only defined between "direct aunt and nephew": The

[7] The implementation allows to declare a definition as "#GENERIC", so that its reference *must* be rewritten in each import.

[8] E.g. in the context of our LaTeX-like d2d_gp architecture, an instantiation of the general purpose article for a particular technical documentation will distribute the required additional in-line elements, like technical terms, abbreviations, symbols, trade marks, formatting styles, not only to the flow text of paragraphs, but also into the content models of captions of tables and figures, into titles, hypertext anchors, footnotes, etc. Whenever further modification of the resulting top level format is necessary, all these now correctly instantiated substructures shall also be immediately addressable for reuse and rearrangement. This is achieved by implicit re-export.

$$
\begin{array}{rl}
T ::= & R \mid \texttt{@}\,T \mid T\,\texttt{\^{}}\,(\,T\,/\,R\,) \mid \#\texttt{empty} \mid \#\texttt{none} \\
\mid & (\,T\,) \mid T\,\texttt{,}\,T \mid T\,\&\,T \mid T\,|\,T \mid T\,\texttt{?} \mid T\,\star \mid T\,\texttt{+} \\
\mid & T\,\tilde{}\,T \mid T\,\tilde{}\,\star \mid T\,\tilde{}\,\texttt{+} \mid [\,\mathbb{I}_T\;T\,] \mid (\texttt{>}T\,) \\
\mid & \texttt{'}\,\mathit{chars}\,\texttt{'} \mid \texttt{"}\,\mathit{chars}\,\texttt{"} \mid \texttt{0x}\mathit{hhhh} \\
\mid & T \cup T \mid T \mathbb{A} T \mid T - T \mid T\,\texttt{..}\,T
\end{array}
$$

$\mathbb{I}_I$ // *ids used as import key. Examples use "*$\texttt{K}$*", "*$\texttt{L}$*", etc.*
$\mathbb{I}_A$ // *ids used as module source location. Examples use "*$\texttt{my\_mod}$*", etc.*
$\mathbb{I}_T$ // *names and tags of definitions. Examples use "*$\texttt{a}$*", "*$\texttt{b}$*", etc.*
$R = \mathbb{I}_I{}^\star \times \mathbb{I}_T$ // *references, relative to the containing module and its imports*
Ξ // *many additional attributes: representation, docu, XSLT, etc.*

$$
\begin{array}{lll}
\mathit{Module} & \triangleq & [\mathit{imps} : \mathbb{I}_I \rightharpoonup \mathit{Import} \;;\; \mathit{defs} : \mathbb{I}_T \rightharpoonup \mathit{Definition}\,] \\
\mathit{Import} & \triangleq & [\mathit{addr} : \mathbb{I}_A\,; \\
 & & \quad \mathit{globalRews} : R \rightharpoonup T\,; \mathit{localRews} : \mathbb{I}_T \rightharpoonup R \rightharpoonup T\,; \mathit{impRews} : \mathbb{I}_I \rightharpoonup \mathbb{I}_I\,] \\
\mathit{Definition} & \triangleq & [\mathit{tag} : \mathbb{I}_T \;;\; \mathit{regExp} : T \;;\; \mathit{repr} : \Xi\,]
\end{array}
$$

// *global constants, fixed per run:*
$\mathsf{module} : \mathbb{I}_A \rightharpoonup \mathit{Module} \qquad \mathsf{top} : \mathbb{I}_A \times \mathbb{I}_T$

$$
\begin{array}{rlrl}
\mathit{import} & : \mathbb{I}_I{}^* \rightharpoonup \mathit{Import} & \mathit{import}(\langle\rangle) &= (\mathsf{top}.1, \{\}, \{\}, \{\}) \\
\mathsf{module}' & : \mathbb{I}_I{}^\star \rightharpoonup \mathit{Module} & \mathsf{module}'(i) &= \mathsf{module}(\mathit{import}(i).\mathit{addr}) \\
\mathsf{def} & : R \rightharpoonup \mathit{Definition} & \mathsf{def}(i,t) &= \mathsf{module}'(i).\mathit{defs}(t) \\
\mathit{base} & : \mathbb{I}^\star \to \mathbb{I}^\star & \mathit{base}(J \blacktriangleleft j) &= J
\end{array}
$$

$$
\frac{\mathit{import}(\pi \blacktriangleleft p) = (a, \_, \_, i)}{\mathit{import}(\pi \blacktriangleleft p \blacktriangleleft q) = \begin{cases} \mathit{import}(\pi \blacktriangleleft q') & \textbf{if}\,(q \mapsto q') \in i \\ \mathsf{module}(a).\mathit{imps}(q) & \textbf{otherwise} \end{cases}}
$$

Table 1. Data types of the underlying domain language; global constants; auxiliary navigation functions.

common ancestor imports two modules and replaces in the first one (in the example "$\texttt{M}$") one of its internal module imports ("$\texttt{H}$") by its own second import ("$\texttt{N}$", see line 4).[9]

3 Algebraic Use of Rewriting Results

Table 1 shows in its top part the data types which represent the underlying domain language and the module import statements. Its middle part shows the two fundamental constant values: The function module maps source addresses to unparameterized module sources. For each particular evaluation run, one definition is fixed as the root element of the parsed text corpus, given by top as a

[9] This could of course be easily extended to a free compositional device, which is always desirable in the theoretical perspective, but does not seem to make much sense in practice, – it soon gets too confusing.

// Stack of currently active local (=expression level) rewritings:
$K_1 = (R \times T)^\star$
// Stack of contexts of the insertion commands `@` *:*
$K_2 = ((K_1 \times \mathbb{I}_I{}^\star \times \mathbb{I}_T) \cup \{\nabla\})^\star$

$\mathsf{visit}[E] : K_1 \times \mathbb{I}_I{}^\star \times \mathbb{I}_T \times K_2 \times T \times E \to (E \cup \{\texttt{Error}\ldots\})$
$\mathsf{action}_\square[E] : \mathbb{I}_I{}^\star \times \mathbb{I}_T \to (E \cup \{\texttt{Error}\ldots\})$

$\mathsf{visit}(k, J, d, m, r_1\ \texttt{\^{}}\ (\ r_2\ /\ j\ )\ , e) = \mathsf{visit}(k \blacktriangleleft (j \mapsto r_2), J, d, m, r_1, e)$
$\mathsf{visit}(k, J, d, m, \texttt{@}\ x, e) = \mathsf{visit}(k, J, d, (m \blacktriangleleft (k, J, d) \blacktriangleleft \nabla)\ , x, e)$
$\mathsf{visit}(k \blacktriangleleft ((i', t') \mapsto x), J, d, m, (i, t) : R, e) = \begin{cases} \mathsf{visit}(k, J, d, m, x, e) & \textbf{if} (i, t) = (i', t') \\ \mathsf{visit}(k, J, d, m, (i, t), e) & \textbf{otherwise} \end{cases}$
$\mathsf{visit}(\langle\rangle, J, d, m, (i, t), e) = \mathsf{visit}(\langle\rangle, \mathit{base}(J), \bot, m, x, e)$
$\quad$ **if** $((i, t) \mapsto x) \in (\mathit{import}(J).\mathit{globalRews} \oplus \mathit{import}(J).\mathit{localRews}(d))$
otherwise :
$\mathsf{visit}(\langle\rangle, J, \_, m \blacktriangleleft \nabla, (i, t), e) = \mathsf{visit}(\langle\rangle, J \frown i, t, m, \mathsf{def}(J \frown i, t).\mathit{regExp}, e)$
$\mathsf{visit}(\langle\rangle, J, \_, m \blacktriangleleft (k', J', d'), (i, t), e) = \mathsf{visit}(k', J', d', m, (i, t), e)$
$\mathsf{visit}(\langle\rangle, J, \_, \langle\rangle, (i, t), e) = \mathsf{action}(J \frown i, t, e)$

$$\boxed{\mathsf{action}_A(i, t, e)} = \begin{cases} X & \textbf{if } X = \mathsf{check}(i, t) \neq \mathit{true} \\ \mathsf{visit}(\langle\rangle, i, t, \langle\rangle, \mathsf{def}(i, t).\mathit{regExp}, e) & \textbf{otherwise} \end{cases}$$

$$\mathsf{check}(i, t) = \begin{cases} \texttt{Error, no import key } i & \textbf{if } i \notin \mathsf{dom}\ \mathit{import} \\ \texttt{Error, no module at } a & \textbf{if } a = \mathit{import}(i).\mathit{addr} \notin \mathsf{dom}\ \mathsf{module} \\ \texttt{Error, no definition at } (a, t) & \textbf{if } t \notin \mathsf{dom}(\mathsf{module}'(i).\mathit{defs}) \\ \mathit{true} & \textbf{otherwise} \end{cases}$$

$$\frac{\square \in \{\ \texttt{,}\ ,\ \texttt{|}\ ,\ \texttt{\&}\ ,\ \texttt{\~{}}\ ,\ \texttt{U}\ ,\ \texttt{A}\ ,\ \texttt{-}\ ,\ \texttt{..}\ \} \qquad \mathsf{visit}(k, J, d, m, r_1, e) = e_1}{\mathsf{visit}(k, J, d, m, r_1\ \square\ r_2, e) = \mathsf{visit}(k, J, d, m, r_2, e_1)}$$

$$\frac{\square \in \{\ \star\ ,\ +\ ,\ ?\ ,\ \texttt{\~{}}\star\ ,\ \texttt{\~{}}+\ \} \qquad t \in \mathbb{I}_T}{\begin{array}{c}\mathsf{visit}(k, J, d, m, r\ \square, e) = \mathsf{visit}(k, J, d, m, \ [\ t\ r\ ]\ , e) \\ = \mathsf{visit}(k, J, d, m, >\ r, e) = \mathsf{visit}(k, J, d, m, r, e)\end{array}}$$

$$\frac{\gamma \in \{\ \texttt{"}\ \alpha\ \texttt{"}\ ,\ \texttt{'}\ \alpha\ \texttt{'}\ ,\ 0\texttt{x}\mathit{hhhh}, \#\texttt{empty}, \#\texttt{none}\}}{\mathsf{visit}(k, J, d, m, \gamma, e) = e}$$

Table 2. Rewriting of source definitions with mere algebraic semantics.

static module's source address plus an identifier of a definition therein. The indicated module is instantiated "as is", without outer rewritings, and the indicated definition serves as the starting point for parsing and addressing.

Its lower part shows some auxiliary navigation functions: *import* translates a sequence of import keys (like "`K.L.M`") into an import statement (contained in some particular module source); module' into the source module mentioned there; and def does the same for a definition.

In the expression language T from Table 1 only the first line is directly related to rewriting and parametrization, the others serve for illustrating the application context: The construct $@\,x$ means the *insertion* or *flattening* of the content models of all definitions referred to in x. This mechanism allows the same definition to be used as XML element content model, or only as a constant with a regular expressions as its value, or in both roles.

The construct e `^ (` f `/` r `)` is an explicit rewriting on expression level, of all identifiers r in the expression e by the replacement expression f. This is useful especially in combination with the insertion operator, as in

```
(@a,@b) ^ ( (x,y*) / c )
```

or as a shorthand notation for repetition, as in

```
(x,x,x) ^ ( (a,(b|c)*,d) / x )
```

`#empty` is a constant which in the parsing process matches the empty input, i.e. which always matches, but delivers no XML output, while `#none` never matches. They are useful in replacement expressions to eliminate parts of sequences or alternatives.

Static we call all facts which follow from the source text of each module alone, per se. *Dynamic* are those which arise after selecting top, by following all imports and references from this starting point. An instantiated module is indicated by its *dynamic path*, i.e. the sequence of import keys from $\mathbb{I}_I{}^\star$ by which it is reachable from top. An instantiated definition in this module is identified by this dynamic path plus its identifier from $\mathbb{I}_T$.[10]

The expansion of definitions, i.e. the application of the rewriting rules is implemented most naturally and in a compact way by the well-known *visitor pattern*: By calling $\mathsf{visit}(\langle\rangle, \langle\rangle, \bot, \langle\rangle, (\langle\rangle, \mathsf{top}.2), \_)$, all reachable definitions are expanded for further processing by the function $\mathsf{visit}(k, J, d, m, x, e)$ from Table 2. In this function signature, the sources of rewrite rules appear with decreasing priority: k is a storage for local rewritings on expression level; J is the dynamic path of the containing instantiated module; d is the identifier of the currently visited definition, or $\bot$ if x lives on module level (i.e. in a replacement expression of an import clause); x is the expression to expand and visit. The rule for $\mathsf{visit}(\ldots, x = (i, t) : R, \ldots)$ processes relative, source text level references. The

[10] The mere data type $\mathbb{I}_I{}^\star \times \mathbb{I}_T = R$ is also used for *relative* references on source level. The transition between both data types is shown in the call to $\mathsf{action}_\square(J \frown i, \ldots)$ in Table 2. The collection of dynamic paths is infinite when circular module imports occur. This will not affect the termination of our algorithms, as discussed in Section 4.3.

first three cases in its definition look for the first applicable rewriting in the accumulated values k, J and m.

The argument e of the generic parameter type E is some additional accumulator parameter fed through all visits, easily implemented as additional visitor field variables. It is not needed here, in the pure algebraic use case, and simply passed through, but will be employed in the context of the next section.

A tricky point is that all rewritings of identifiers shall be *applied to insertions twice*, before and after expansion. This is again a mere pragmatic design decision: Both modes of operation are needed. Assume a definition like "`tags article = ..., @ table, ...`", i.e. the definition of the contents of an element called "`article`" which includes the spliced contents of the element "`table`". Then these two rewritings are both sensible:

- "use the content model of `article`, replacing (all references to and) all insertions of `table` by those of `mytable`;"
- "use `article`, but in the inserted contents of `table` replace `caption` by `#empty`."

It seems that separate language elements for both modes either become very complicated or must give up compositionality. Furthermore, we did not find a use case which required to execute a replacement in one phase, and to suppress it explicitly in the other. So we could unify both meanings and keep the language front-end small. The command for splicing ∇ and the context for rewriting of the spliced contents can be stored to the same stack $m : K_2$.

For the placeholder $\mathsf{action}_\square()$ the intended functionality must be plugged in, e.g. the function $\mathsf{action}_A()$ which performs just a context check. In the production context of course different functions are plugged in which realize the intended semantics.[11] The last three rules in Table 2 simply descend into the expressions of the underlying domain language, are thus specific and must of course also be treated individually by any derived production visitor.

The collection of function definitions from Table 2 is complete, covers common static error cases as detected by $\mathsf{check}()$ and thus defines a precise operational semantics for the rewriting process. This is totally independent from further properties of the underlying domain language, here: T.

Two problems do not arise: First, cycles and thus infinite structures are not an issue, because the parsing process matches them against a *finite* input, or at least against an always finite prefix of the input, which limits all access operations in a natural way.

Second, duplications are not an issue: As usual in evaluation systems of this kind, only the algebraic semantics are exploited. One and the same definition may be unfolded and instantiated arbitrarily often, because the effect of all these copies will implode again when the result, here: the parse tree, is finally delivered.

[11] In our case, which is parsing, the current head of the input token stream is tested, possibly consumed, a new XML result element is possibly opened or closed, etc.

$$
\begin{aligned}
E_R &= R \rightharpoonup {\mathbb{I}_I}^\star \\
\mathsf{cmpDef} &: {\mathbb{I}_I}^\star \times {\mathbb{I}_I}^\star \times \mathbb{I}_T \times E_R \to (E_R \times \mathit{boolean}) \\
\mathsf{cmpExp} &: T \times T \times {\mathbb{I}_I}^\star \times {\mathbb{I}_I}^\star \times \mathbb{F}\, R \times E_R \to (E_R \times \mathit{boolean}) \\
\mathsf{refs} &: T \to \mathbb{F}\, R \;\; /\!/ \; \textit{delivers all references appearing in an expression.} \\
\mathit{sourceAddr} &: R \rightharpoonup (\mathbb{I}_A \times \mathbb{I}_T) \\
\mathit{sourceAddr}(i,t) &= (\mathit{import}(i).\mathit{addr}, t)
\end{aligned}
$$

$$
\frac{Q = \mathit{sourceAddr}^{-1}(\mathit{sourceAddr}(i,t)) \triangleleft e}{
\begin{aligned}
\mathsf{action}_B((i,t),e) &= \begin{cases} X & \textbf{if } X = \mathsf{check}(i,t) \neq \mathit{true} \\ ((j,t),e) & \textbf{if } ((i,t) \mapsto j) \in Q \\ \mathsf{ta}(e,(i,t),\mathsf{ran}\, Q) & \textbf{otherwise} \end{cases} \\
\mathsf{ta}(e,(i,t),\{\}) &= \mathsf{action}_A((i,t), e \cup \{(i,t) \mapsto i\}) \\
\mathsf{ta}(e,(i,t),\{j\} \cup \sigma) &= \begin{cases} ((j,t),e') & \textbf{if } \mathsf{cmpDef}(i,j,t,e) = (e',\mathit{true}) \\ \mathsf{ta}(e,(i,t),\sigma) & \textbf{otherwise} \end{cases}
\end{aligned}
} \;\textbf{(prev)}
$$

$$
\frac{
\begin{gathered}
V = \mathsf{refs}(\mathsf{def}(i,t).\mathit{regExp}) \\
W_\zeta = V \triangleleft (\mathit{import}(\zeta).\mathit{globalRews} \oplus \mathit{import}(\zeta).\mathit{localRews}(t)) \\
\overline{V_\zeta} = (\mathsf{dom}\, \mathit{import}(\zeta).\mathit{globalRews} \setminus V) \cup \bigcup x \neq t \bullet \mathsf{dom}\, \mathit{import}(\zeta).\mathit{localRews}(x) \\
\mathsf{tb}(e \cup \{(i,t) \mapsto j\}, i, j, W_i, W_j) = (e', X')
\end{gathered}
}{\mathsf{cmpDef}(i,j,t,e)} \;\textbf{(defs)}
$$

$$
= \begin{cases}
(e, \mathit{false}) & \textbf{if } \mathit{sourceAddr}(i,t) \neq \mathit{sourceAddr}(j,t) \\
(e, j \stackrel{?}{=} x) & \textbf{elseif } ((i,t) \mapsto x) \in e \\
(e, \mathit{false}) & \textbf{elseif } \mathsf{dom}\, W_i \neq \mathsf{dom}\, W_j \\
(e, \mathit{false}) & \textbf{elseif } X' = \mathit{false} \\
(e', \mathit{true}) & \textbf{elseif } \overline{V_i} = \overline{V_j} = \{\} \\
\mathsf{cmpExp}(\mathsf{def}(i,t).\mathit{regExp}, \mathsf{def}(j,t).\mathit{regExp}, i, j, \mathsf{dom}\, W_i, e') & \textbf{otherwise}
\end{cases}
$$

$$
\begin{gathered}
\mathsf{tb}(e,i,j,\{\},\{\}) = (e, \mathit{true}) \\
\mathsf{tb}(e,i,j,\{r \mapsto s_1\} \cup \sigma_1, \{r \mapsto s_2\} \cup \sigma_2) \\
= \begin{cases} \mathsf{tb}(e',i,j,\sigma_1,\sigma_2) & \textbf{if } \mathsf{cmpExp}(s_1,s_2,i,j,\{\},e) = (e',\mathit{true}) \\ (e, \mathit{false}) & \textbf{otherwise} \end{cases}
\end{gathered}
$$

$$
\frac{\mathsf{cmpExp}(s_1,s_2,I,J,N,e) = (e',\mathit{true})}{\mathsf{cmpExp}(r_1\ \hat{}\ (\ s_1\ /\ t_1\ )\ , r_2\ \hat{}\ (\ s_2\ /\ t_2\ )\ , I, J, N, e)} \;\textbf{(subst)}
$$

$$
= \begin{cases} (e, \mathit{false}) & \textbf{if } t_1 \neq t_2 \\ \mathsf{cmpExp}(r_1, r_2, I, J, N \cup \{t_1\}, e') & \textbf{otherwise} \end{cases}
$$

$$
\frac{i = I \frown i' \quad j = J \frown j'}{\mathsf{cmpExp}((i',t),(j',u),I,J,N,e)} \;\textbf{(exp0)}
$$

$$
= \begin{cases}
(e, \mathit{false}) & \textbf{if } t \neq u \\
(e, \mathit{true}) & \textbf{elseif } (i',t) \in N \\
(e, \mathit{false}) & \textbf{elseif } \mathit{sourceAddr}(i,t) \neq \mathit{sourceAddr}(j,t) \\
\mathsf{cmpExp}((i',t),(j',t),I,J,N,e \cup \{(j,t) \mapsto j\}) & \\
\qquad & \textbf{elseif } j \notin \mathsf{dom}\, e \\
(e, j \stackrel{?}{=} x) & \textbf{elseif } ((i,t) \mapsto x) \in e \\
\mathsf{cmpDef}(i,j,t,e \cup \{(i,t) \mapsto j\}) & \textbf{otherwise}
\end{cases}
$$

Table 3. Reification I, testing for equivalence of rewriting results

$$\frac{\begin{array}{c}\nabla_{\_} \in \{\,\star\,,\,+\,,\,?\,,\,\tilde{}\,\star\,,\,\tilde{}\,+\,\} \\ c_{\_} \in \{\,\texttt{"}\,\alpha\,\texttt{"}\,,\,\texttt{'}\,\alpha\,\texttt{'}\,,\,0\texttt{x}\,hhhh,\#\texttt{empty},\#\texttt{none}\}\end{array}}{\mathsf{cmpExp}(c_1,c_2,\_,\_,\_,e) = (e, c_1 \stackrel{?}{=} c_2)}\ \textbf{(exp1)}$$

$$\mathsf{cmpExp}(r_1\,\nabla_1, r_2\,\nabla_2, I, J, N, e) = \begin{cases}\mathsf{cmpExp}(r_1, r_2, I, J, N, e) & \textbf{if } \nabla_1 = \nabla_2 \\ (e, \mathit{false}) & \textbf{otherwise}\end{cases}$$

$$\mathsf{cmpExp}(\,>\ r_1,\,>\ r_2, I, J, N, e) = \mathsf{cmpExp}(\,\texttt{@}\ r_1,\,\texttt{@}\ r_2, I, J, N, e) = \mathsf{cmpExp}(r_1, r_2, I, J, N, e)$$

$$\mathsf{cmpExp}(\,[\ t_1\ r_1\ ]\,,\,[\ t_2\ r_2\ ]\,, I, J, N, e) = \begin{cases}\mathsf{cmpExp}(r_1, r_2, I, J, N, e) & \textbf{if } t_1 = t_2 \\ (e, \mathit{false}) & \textbf{otherwise}\end{cases}$$

$$\frac{\begin{array}{c}\square_{\_} \in \{\,\texttt{,}\ \ ,\,\texttt{|}\,,\,\texttt{\&}\,,\,\tilde{}\,,\,\texttt{U}\,,\,\texttt{A}\,,\,-\,,\,\texttt{..}\,\} \\ \mathsf{cmpExp}(r_1, r_2, I, J, N, e) = (e', X)\end{array}}{\mathsf{cmpExp}(r_1 \square s_1, r_2 \square s_2, I, J, N, e)}\ \textbf{(exp2)}$$

$$= \begin{cases}(e, \mathit{false}) & \textbf{if } \square_1 \neq \square_2 \\ (e, \mathit{false}) & \textbf{elseif } X = \mathit{false} \\ \mathsf{cmpExp}(s_1, s_2, I, J, N, e') & \textbf{otherwise}\end{cases}$$

$$\boxed{\mathsf{action}_C((i,t),e) = \mathsf{action}_A(\,(e(i,t),t),\,e)}$$

Table 4. Reification II, comparison of expressions, specific for the underlying domain language, and finally resulting visitor for both cases.

4 Reified Use of Rewriting Results

The situation changes fundamentally as soon as the expanded definitions shall be "reified" and treated as identifiable objects, as it is frequently necessary in practice for error diagnosis, user documentation, collection of statistic data, etc.

The current implementation of d2d administers polyglot documentation for every XML element of a text structure in form of a hyper text, using d2d recursively to document itself. These texts naturally are defined for the unexpanded sources, and a computer scientist or language expert can possibly profit and "link" them mentally. But the addressed domain experts are much more helped with a documentation of the *fully expanded* structure definition, which tells them directly where certain tags are allowed, required, forbidden, etc., and what their role is in this particular context.

In all these cases, in contrast to the expansion process as described in the preceding section, two severe issues arise: (A) the expansion is not limited by the input data, therefore infinite structures *must* be eliminated explicitly, and (B) multiple equivalent expansions of the same source definition *should* not occur, for not confusing the user. Consequently, the properties of the underlying domain language, here: T, can no longer be totally abstracted from, as it was possible in the preceding section. The finiteness of the recursion depends on the definition

of equality, and any notion of equivalence is related to the intended semantics and to the pragmatic requirements from a user's point of view.

Nevertheless, this influence is limited and can be isolated. For this purpose, the following algorithm is given as a framework of two layers: The upper one steps through definitions, using an optimized strategy, and is presented in Table 3. It is independent of the properties of the underlying domain language. These come into play on the lower level, which compares expressions, see Table 4.

4.1 Comparing Definitions

Problem (B) from the list above is easier to deal with, and a typical unification problem. We search for the greatest fixpoint: All different instantiations of the same source text definition for which no counterexample is found are put into the same equivalence class. A counterexample is a sub-structure which shows a difference. These equivalence classes are constructed by recursive descent, similar to the visiting process above, enriched by very simple backtracking. In contrast to the algebraic case, here two (2) definitions or expressions are visited in parallel. During this descent, hypotheses of equivalence are collected.

This process is simplified significantly by two properties of the overall equation system: First, there is no negation, i.e. the addition of hypotheses is monotone. Second, there are not many alternatives, i.e. whenever the hypothesis of two instantiations being equivalent is added, there is only one single way to prove it, namely to show the equivalence of their complete instantiated substructures in a one-to-one fashion – there are no choice points with a finer granularity than complete definition instantiations and their equivalence class, save those induced by properties of the underlying domain language, see next section.

The definition re-uses the visitor code from the preceding section: $\mathsf{action}_\square$ in Table 2 is set to to action_B from Table 3, and then top is visited.[12] The new code makes now use of the accumulator parameter e to hold a map of type E_R which at the end of the process for each reference value reachable from top will give an equivalence class, represented by one of its members. Only this representative will appear in the generated documentation, statistics, etc.

When an absolute reference (i, t) is reached by the visitor code, i.e. when $\mathsf{action}_B((i, t), e)$ is called, it is checked by the rule **(prev)** whether the same source has been visited previously. If not, this instantiation starts its own equivalence class. Else it is checked whether this instantiation has already been classified, or else, whether it is equivalent to one of the already established equivalence

[12] In contrast to the parsing job as described above, in the practical use cases covered here it is often sensible to have more than one root symbol to start with. E.g., in the context of `d2d_gp`, the documentation for "`deliverables:article`", "`deliverables:book`" and 'deliverables:webpage" should be generated and presented to the user together, in one turn, because these share most of their substructures. In our model this is achieved by visiting a synthetic definition of a corresponding disjunction. Furthermore, the expansion of the "`@`" operator as implemented above is no longer required, but optional.

classes. These are stepped through by the auxiliary function ta(). For this purpose we define:

- Two expressions are equivalent iff their structure is identical, and all references go to instantiations of definitions which are again equivalent. This is implemented by **(subst)** and **(exp0)** to **(exp2)**.
- The equivalence of two definitions is tested by **(defs)** and implies
  (a) they have the same original source, and
  (b) the set of references affected by rewrite rules ($\mathsf{dom}\, V_{i/j}$) is the same for both, and
  (c) the replacement expressions of these effective rewrite rules are equivalent, which is tested by the function tb(), and finally
  (d) the rewritten content models of both definitions are equivalent.

The function $\mathsf{cmpDef}(i, j, t, e)$ checks whether the definitions with the identifier t in the module instantiations reachable by the dynamic paths i and j can be made equivalent under the hypotheses in e by adding additional hypotheses.

Please note that condition (b) implies that we do not look for definitions which are "accidentally" identical, e.g. by rewriting a particular reference in one definition to itself, while not touching it in the other, with the same result, as in "`(a,b)^a/a`" vs. "`(a,b)`". This strategy seems more adequate to the ergonomic needs of the user: Different sources of parametrization are always "on purpose", related to some need for structuring, which shall be reflected in documentation etc. anyhow.

W.r.t. execution complexity it holds that all expressions in **(defs)** except the two which call tb() and cmpExp() have *static* values, or require only a simple map look-up in e. Comparing the replacement expressions of the rewrite parameters is an optimization instead of comparing the two resulting rewritten content models completely: Since the sources are identical, differences only come from the replacements.

Nevertheless, checking of condition (d) may be necessary, additionally. This is due to all rewrite rules which are *not* in $V_{i/j}$, which are *not directly* applied to the currently visited definitions, but to other definitions in the containing modules. The current definitions refer to these others in very different forms, directly, or indirectly via further definitions, or even via replacement expressions in further module imports. These effects may differ in both instantiations. The test of this condition is expensive, therefore it is done at last. It requires the descent into both content models, and recursively to all definitions referred therein, up to the first counterexample found. Otherwise it will further develop the state of e. But condition (d) is definitely true if both considered import clauses do not carry more rewrite rules beyond $V_{i/j}$, the directly effective ones; this inference speeds up the comparison substantially, see the test for $\overline{V_i} = \overline{V_j} = \{\}$.

When there is no instantiation to which the visited instantiation is equivalent, then action_A must be called to descend into all reachable definitions, for their classification. This is always the case when a particular definition source is visited for the very first time.

4.2 Comparing Expressions

The function $\mathsf{cmpExp}(r_1, r_2, I, J, N, e)$ checks whether the two expressions r_1 and r_2, to be evaluated in the instantiated modules given by the dynamic paths I and J respectively, are equivalent under the hypotheses in e. In case of success additional hypotheses will possibly be added to e. The set N contains those references which need not be checked: They are affected by rewrite rules and their replacement expressions have already been compared successfully when testing condition (c) from above. The function is applied for testing criterion (c) or (d) from the list above; it is initially called from **(defs)**, and then recursively.

If the constructors of both expressions differ, i.e. no rule in Table 4 matches, the unification fails by an implicit default rule, which indicates a *structural difference* of constructor application. For most underlying domain languages this fact and its effect are independent of any rewriting of references, only related to the involved components of the source text, and thus can be cached in some additional global monotone storage. But this is not necessarily the case. In case of a language similar to d2d it may be more appropriate to treat structural differences as context-sensitive and healable: in T the expressions "`(a,b)^(#empty/b)`" and "`(a|b)^(#none/b)`" could be considered equivalent.

The parameters of the matching rules are not symmetrical: All references occurring in r_1 are already classified, those in r_2 are possibly free for unification. All resulting hypotheses are stored to e and checked for consistency; any conflict will make the unification fail and the calling level will unwind the hypotheses by recurring to some pre-state of e. This simple procedure is feasible because there are no further choice points after the most recent hypothesis has been set by **(defs)** for a definition instantiation.

But on the level of expression alternatives may arise for equivalences, when the intended semantics of the underlying domain language are taken into account, e.g. coming from associative and commutative constructors. These must be modeled as additional inference rules in Table 4 accordingly. The d2d implementation eliminates the need for these further choice points by transforming expressions into a normal form, based on a complete order of constructors and primitive scalars.

The version action_B cannot be used for the original, algebraic purpose anymore, because it cuts all recursions on the object level by simply storing an hypothesis into e. This is feasible because the mere recursion in the expression values does not detect further dependencies from rewriting parameters, – these are always found in the first unrolling.

But once the map $e : E_R$ has been constructed, the simple variant action_C is plugged in the visitor of Table 2 and visit() may be called, parametrized accordingly, including this e.[13] The resulting visitor process combines with equal rights algebraic use, here: control of a parsing process, and reified use, like statistics, documentation or user help functions: Only one and the same representative per equivalence group will always be visited.

[13] Additionally, the call to check() may be skipped.

4.3 Termination

The above-mentioned question (A) of termination is much harder to deal with. It is only related to cyclic imports of modules, since cyclic references in the expression language can be safely ignored, as described above. A *sufficient* condition for termination is that every reference which appears directly or indirectly in a replacement expression for itself, due to a circular import, does so without any constructor application from T, i.e. is totally *identical* with this expression, modulo all renamings occurring on the circle. This condition prevents the import parameter from accumulating complexity with every turn of the unrolling of the import cycle, i.e. growing beyond all limits. Then our algorithm obviously terminates: There are statically only finitely many rewritten parameters and replacement expressions; all recursion goes through the functions `ta`() and `tb`() and both are consumptive by always removing one element from some finite set valued parameter. Less restrictive conditions for termination must take into account the expression structure of the underlying domain language and are subject of future work.

5 Related Work

The problem of rewriting in general is an important discipline on its own, with a wide range of theories, tools and applications. Nearly all of them differ from our approach since they deal with automated transformations, or preservations of properties, or semantical identity, etc. Contrarily, in our context rewriting is a mere practical means to derive new structures which are intended not to be equivalent.

There are only few cases when rewriting has been applied to grammars: Already in 2001 Lämmel described a collection of grammar transforming operators and their properties [7]. The motivating context is, similar to ours, a practical one, namely to reconstruct the originally intended semantics of ill-formed artefacts. Cyclic definitions and reification are not an issue. His operators and combinators live one and two levels above our simple substitution, respectively, and their influence on properties is exhaustively discussed. It seems promising to integrate some of these results into our future work; same holds for the fundamental considerations in [8].

RelaxNG [2] has an module import mechanism based on a notion of "file", which is defined by URLs. Thus mechanism supports replacement of definitions. The file contents is inserted on the front-end level, and the general incremental definition operators can be applied, not regarding the provenience. Our approach differs substantially: Since we support replacement of references, (1) the original definitions stay accessible, and (2) the rewriting takes place for a whole group of definitions in a consistent way.

The W3C XSD schema language has methods for *type derivation*. [9] Derived types can be used to define "substitution groups" which allow to replace one element class by another. But since type derivation can both extend and restrict types, the construction of precise semantics seems infeasible.

References

1. Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler, Fran cois Yergeau, and John Cowan. *Extensible Markup Language (XML) 1.1 (Second Edition)*. W3C, `http://www.w3.org/TR/2006/REC-xml11-20060816/`, 2006.
2. Clark and Murata. *Document Schema Definition Language (DSDL) – Part 2: Regular-grammar-based validation – RELAX NG*. ISO/IEC, `http://standards.iso.org/ittf/PubliclyAvailableStandards/c052348_ISO_IEC_19757-2_2008(E).zip`, 2008.
3. Donald E. Knuth. *The TEXbook*. Addison-Wesley, 1987.
4. Leslie Lamport. *LaTeX User's Guide and Document Reference Manual*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.
5. Markus Lepper and Baltasar Trancón y Widemann. d2d — a robust front-end for prototyping, authoring and maintaining XML encoded documents by domain experts. In Joaquim Filipe and J.G.Dietz, editors, *Proceedings of the International Conference on Knowledge Engineering and Ontology Delelopgment, KEOD 2011*, pages 449–456, Lisboa, 2011. SciTePress. `http://bandm.eu/d2d/d2d2011_4.pdf`.
6. Markus Lepper, Baltasar Trancón y Widemann, and Jacob Wieland. Minimze mark-up ! – Natural writing should guide the design of textual modeling frontends. In *Conceptual Modeling — ER2001*, volume 2224 of *LNCS*. Springer, November 2001.
7. Ralf Lämmel. Grammar adaptation. In *PROC. FORMAL METHODS EUROPE (FME) 2001, VOLUME 2021 OF LNCS*, pages 550–570. Springer-Verlag, 2001.
8. Ralf Lämmel and Wolfgang Lohmann. Format evolution, 2001.
9. Schema Working Group. *XML Schema*. W3C Candidate Recommendation, `http://www.w3.org/XML/Schema`, 2004.
10. J.M. Spivey. *The Z Notation: a reference manual*. Prentice Hall, 1988.

A Mathematical Notation

The employed mathematical notation is fairly standard, inspired by the Z notation [10]. But for leaner notation, we add some overloading. The following table lists some details:

$\mathbb{F}\ A$	Finite power set, the type of all *finite* subsets of the set A.
$A \times B$	The product type of two sets A and B, i.e. all pairs $\{c = (a, b) \mid a \in A \wedge b \in B\}$. We write $c.1$ and $c.2$ for the components a and b.
$A \to B$	The type of the *total* functions from A to B.
$A \nrightarrow B$	The type of the *partial* functions from A to B.
$A \leftrightarrow B$	The type of the relations between A and B.
$\mathsf{dom}\, a, \mathsf{ran}\, a$	Domain and range of a function or relation.
$r \fatsemi s$	The composition of two relations: The smallest relation s.t. $a\ r\ b \wedge b\ s\ c \Rightarrow a\ (r \fatsemi s)\ c$
$r \oplus s$	Overriding of function or relation r by s. Pairs from r are shadowed by pairs from s: $r \oplus s = \big(r \setminus (\mathsf{dom}\, s \times \mathsf{ran}\, r)\big) \cup s$
$r \oplus (a \mapsto b)$	Overriding of function or relation r by a single maplet.
$S \lhd R$	$= R \cap (S \times \mathsf{ran}\, R)$, i.e. domain restriction of a relation.
r^{-1}	The inverse of a relation
$a \blacktriangleright \beta$	A sequence seen as a first element a and the rest sequence β.
$\alpha \frown \beta$	Concatenation of two lists.
$\alpha \blacktriangleleft b$	A list(/stack) with element b preceded by a prefix α.
$T \triangleq [a : A; b : B]$	Definition of a schema, i.e. a product type with named projections. Equivalent to $T = A \times B$, with $\pi_1 = .a$ and $\pi_2 = .b$
$a \stackrel{?}{=} b$	Function delivering the Boolean value which reflects equality.

Functions are considered as special relations, i.e. sets of pairs, like in "$f \cup g$".