Parsing and Printing of and with Triples

Sebastiaan J. C. Joosten

Computational Logic Group, UIBK Innsbruck
Email: Sebastiaan.Joosten@uibk.ac.at

Abstract. We introduce the tool Amperspiegel, which uses triple graphs for parsing, printing and manipulating data. We show how to conveniently encode parsers, graph manipulation-rules, and printers using several relations. As such, parsers, rules and printers are all encoded as graphs themselves. This allows us to parse, manipulate and print these parsers, rules and printers within the system. A parser for a context free grammar is graph-encoded with only four relations. The graph manipulation-rules turn out to be especially helpful when parsing. The printers strongly correspond to the parsers, being described using only five relations. The combination of parsers, rules and printers allows us to extract Ampersand source code from ArchiMate XML documents. Amperspiegel was originally developed to aid in the development of Ampersand.

This is the preprint of the article that is part of the Proceedings of RAMICS 2017, Lecture Notes in Computer Science book series (LNCS, volume 10226), published April 2017. Apart from this notice and some layout differences, it is identical (this includes the numbers of figures and definitions, but not the page numbers).

1 Introduction

We introduce a framework for language transformations, called Amperspiegel. We see a language transformation as something that consists of three parts: a parser, a series of semantic transformations, and a printer. To describe these parts and their behaviour, we adopt the view that everything can be described in relations.

Languages are described by encoding a Context Free Grammar in four relations. Transformations are described using a set of declarative rules in a subset of relation algebra. The printing then occurs using the inverse of the parser.

Like the parser, also the transformation and the printer are expressed in relations. Consequently, the framework has some reflective capabilities. The name Amperspiegel stems from the framework's relation to Ampersand [4], while emphasising that it has reflection.[1] It is stand-alone software (`github.com/sjcjoosten/Amperspiegel`), so it can be used in projects other than Ampersand as well. Code specific to this paper can be found at: `cl-informatik.uibk.ac.at/users/sjoosten/as/`

As an example, Section 7 creates a link between two tools: ArchiMate and Ampersand. We show how to parse files that describe a software architecture written in an

[1] Adding to Amperspiegel's reflection are the switches `collect` and `distribute`, which are not described in this paper.

ArchiMate XML file. The structure is transformed, and then printed as a description of the same architecture as an Ampersand ADL file. This is done using Amperspiegel.

The focus of this paper is on the concepts behind Amperspiegel, seen as a stand-alone tool. Section 2 gives an overview of the tool and describes its use. We define a parser, a rule engine, Amperspiegel's embedding of a set of rules, and a printer, in Sections 3, 4, 5 and 6 respectively.

Related work. Several tools combine parsing and printing with transformations, including meta-programming languages such as Rascal [7] and Stratego [2], or programming language workbenches such as Spoofax [6]. Amperspiegel offers a fundamental approach to meta-programming, offering these features with a minimal implementation. Excluding a file that configures the initial state of Amperspiegel, it is under a thousand lines of Haskell code.

To achieve this, Amperspiegel borrows from several best practices. Using a Context Free Grammar for parsing and for printing is done before by Mark van den Brand [1]. Deriving new facts with rules, as Amperspiegel does, is similar to the declarative programming language datalog$\pm$ [3]. Its restriction to triples, in a style like Amperspiegel, is described by Edward Robertson [9]. We have not seen a Context Free Grammar described through relations, and this allows to combine these concepts in a novel way. This makes building source-to-source transformations surprisingly easy and modular.

2 Overview of Amperspiegel

To transform languages, Amperspiegel can parse input, apply rules, produce output, and assemble these components in a single execution. This overview shows how components are assembled. Amperspiegel interprets command-line arguments as commands. They are executed from left to right.

The most important actions are '`apply`', '`parse`' and '`print`'. These actions are performed on structures that correspond to a kind of labelled graph. We refer to these structures as 'graph', and explain how they can be understood as a set of homogeneous relations. This interpretation is important, as we expect the Amperspiegel user to think of these structures as a description through several relations.

Initially, there are pre-defined graphs in Amperspiegel. Some of these graphs represent parsers. Using `parse`, a parser is used to parse an input file, creating another graph. Graphs can be manipulated by rules using `apply`, again creating a graph. A graph can be printed to stdout by `print`.

We illustrate Amperspiegel's command line interface by showing how to execute:

```
ds1 := parse data file1
ds2 := parse rule file2
res := apply ds2 ds1
print data res
```

This example uses built-in parsers and printers to read in some data (in `file1`), apply some transformation to it (given by `file2`) and print the result on stdout. It uses the

same internal parser as a printer, called `data` to both read the data and print the result. The transformation is parsed using an internal parser called `rule`. For this code, Amperspiegel's command-line interface is used as follows:

```
amperspiegel -parse data file1 ds1 -parse rule file2 ds2 \
    -apply ds2 ds1 res -print data res
```

Since Amperspiegel is used to translate a variation of one language into another, a graph can be used in place of the default parser too:
`Amperspiegel -parse cfg path-to/parser mdp -parse mdp my-data ds1`
uses the parser `path-to/parser`, described in `CFG` syntax, to parse the file `my-data` in the new syntax referred to as `mdp`.

Amperspiegel's graph-based notion of data is similar to that used for the semantic web. Another way to view such a graph is as a structure interpreting a set of binary-relation symbols.

Definition 1 (Graph). *A directed labeled graph $G = (\mathcal{L}, V, E)$ is given by a finite set of labels L, a set of vertices V, and a set of edges $E \subseteq \mathcal{L} \times V \times V$.*

In this paper we simply say *graph* when we mean a directed labeled graph. This notion of graph is useful when thinking about the implementation of Amperspiegel. From the perspective of an Amperspiegel user, however, it is more useful to think of this structure as a set of homogeneous binary relations. To help strengthen this way of thinking, we suggestively write $(v, w) \in_G r$ for $(r, v, w) \in E$. Indeed, when the label r occurs in an Amperspiegel script, it is natural to interpret it as a relation symbol. We say that a graph is *finite* if and only if its set of vertices is finite.

There is no way to access the structure of nodes in Amperspiegel, except through the edges in which they occur. Thus, the set of vertices is implicitly equal to those vertices that occur in an edge. In the following sections, we show how a finite graph can describe a parser, a printer and a data-transformation (set of rules).

3 Parsing

To specify parsers, we use Context Free Grammars. While a Context Free Grammar (CFG) is typically used to define a set of strings called 'language', we focus on how CFGs relate to graphs. This section relates CFGs to graphs in two ways: First, a CFG can be used to interpret a string as a parse graph. This allows the Amperspiegel user to read graphs from a file that has a certain file format. Second, a CFG can itself be encoded as a graph. This allows the Amperspiegel user to specify and use its own CFGs.

Definition 2 (Context Free Grammar). *A* CFG *$g = (P, \Sigma, C, S)$ is given by a relation $C \subseteq P \times (P + \Sigma)^*$ and a start symbol $S \in P$, where P denotes the finite set of non-terminals, and Σ denotes the set of terminals. A pair in C is called a production rule.*

We present a CFG by listing C. The set of terminals Σ is disjoint from P, and $S =$ '`S`'. See for instance Example 1. Strings in $(P + \Sigma)^*$ are given by separating elements in $P + \Sigma$ with spaces.

Example 1. $\mathtt{S} \mapsto \mathtt{0\,L\,S}$ $\quad \mathtt{S} \mapsto \varepsilon$ $\quad \mathtt{L} \mapsto \mathtt{S\,1\,L}$ $\quad \mathtt{L} \mapsto \varepsilon$

It follows from convention that P is the two-element set containing S and L, and that Σ contains 0 and 1.

3.1 Obtaining a graph by parsing a string

A CFG (P, Σ, C, S) gives rise to a parser graph $\mathbb{G}$ in which P are the labels, and $\Sigma^* \times P$ are the vertices. This graph is infinite, as it contains all possible parses. It is independent of the start nonterminal S. For a given string s, the parse graph is the subgraph of $\mathbb{G}$ of nodes and edges reachable from the node (s, S), which is guaranteed to be finite. We give an example before the definitions. The empty string is written as ε.

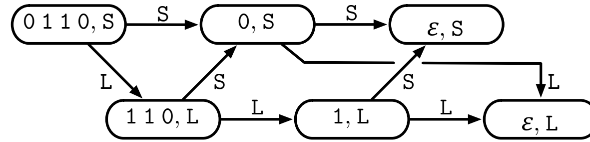

Fig. 1: The parse graph of Example 2

Example 2. For the CFG of Example 1, the parse graph of 0 1 1 0 is given by:

$$((\mathtt{0\,1\,1\,0}, \mathtt{S}), (\mathtt{0}, \mathtt{S})), ((\mathtt{1\,1\,0}, \mathtt{L}), (\mathtt{0}, \mathtt{S})), ((\mathtt{1}, \mathtt{L}), (\varepsilon, \mathtt{S})), ((\mathtt{0}, \mathtt{S}), (\varepsilon, \mathtt{S})) \in_G \mathtt{S}$$
$$((\mathtt{0\,1\,1\,0}, \mathtt{S}), (\mathtt{1\,1\,0}, \mathtt{L})), ((\mathtt{1\,1\,0}, \mathtt{L}), (\mathtt{1}, \mathtt{L})), ((\mathtt{1}, \mathtt{L}), (\varepsilon, \mathtt{L})), ((\mathtt{0}, \mathtt{S}), (\varepsilon, \mathtt{L})) \in_G \mathtt{L}$$

In Example 2, each edge in the parse graph is of the form $((s_1, p), (s_2, p')) \in_{\mathbb{G}} p'$, indicating that s_1 parses as p via a production $(p, \cdots p' \cdots) \in C$, where the substring s_2 parses as p'. A parser graph captures all possible parse graphs, plus edges to terminal symbols that help in our definition of parser graph.

Definition 3 (Parser graph). *Given CFG (P, Σ, C, S), the graph $\mathbb{G} = (P, \Sigma^* \times P, E)$ is the parser graph of (P, Σ, C, S), in which E is the least set of edges such that for each $(p, p_0 \cdots p_n) \in C$, and for every $s = s_0 \cdots s_n \in \Sigma^*$:*

$$\left(\forall i \leq n. \begin{pmatrix} & (\exists x, p'. \; ((s_i, p_i), x) \in_{\mathbb{G}} p') \\ \vee & ((p_i, \varepsilon) \in C \wedge s_i = \varepsilon) \\ \vee & (p_i \in \Sigma \wedge s_i = p_i) \end{pmatrix} \right) \Rightarrow \left(\forall i \leq n. \; ((s, p), (s_i, p_i)) \in_{\mathbb{G}} p_i \right)$$

This formula states that if each of $p_0 \cdots p_n$ can be parsed as a corresponding $s_0 \cdots s_n$, then p can be parsed as $p_0 \cdots p_n$ and corresponding edges exist in $\mathbb{G}$.

Definition 4 (Parse graph). *A parse-graph for the string s and CFG (P, Σ, C, S) is the subgraph of the parser graph $\mathbb{G}$ that is reachable from (s, S) via edges in P.*

A parse graph of s is finite. It contains only vertices (s', v) in which s' is a substring of s and $v \in P$. There are at most $n(n+1)/2 + 1$ substrings in a string of length n, and P is finite. Therefore every parse graph is finite.

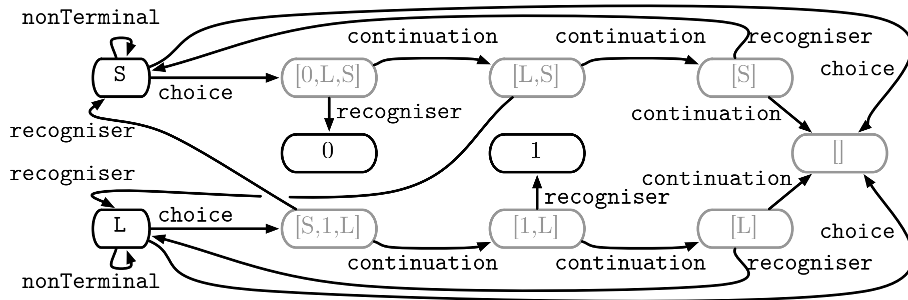

Fig. 2: The CFG of Example 1 drawn as a graph

3.2 Describing a context free grammar with a graph

This section focuses on how CFGs have been implemented in Amperspiegel. We encode a CFG as a graph, allowing a light-weight implementation. This also allows us to express a CFG that can parse its own description and yield the CFG parser itself.

The CFG (P, Σ, C, S) is encoded as a graph $G = (\mathcal{L}, V, E)$, by making C explicit, and using a default element for S. The label `choice` $\in \mathcal{L}$ describes C. Amperspiegel does not have sums or lists as built-in types, so we reconstruct the type of vertices from the labels of edges. The structure of elements of $(P + \Sigma)^*$ is described using three labels: `recogniser`, `continuation` and `nonTerminal`. Amperspiegel uses `recogniser` and `continuation` rather than, say, `head` and `tail`. This choice is less likely to cause name clashes when combining graphs by taking their union, as we will do in Section 7. We combine the edges labelled `choice` with the ones that describe structure in a single graph, so $V = P + \Sigma + (P + \Sigma)^*$. In the sense of Section 4, vertices in $P + \Sigma$ act as constant symbols while vertices in $(P + \Sigma)^*$ act as variable symbols.

For Example 1, the corresponding CFG is given as a graph in Figure 2. Nodes that encode lists in $(P+\Sigma)^*$ are drawn in grey. The lists that make up these nodes are written in Haskell notation to emphasise difference between $\mathtt{S} \in P$ and $[\mathtt{S}] \in (P + \Sigma)^*$.

A CFG (P, Σ, C, S) corresponds to a graph G if:

$$(p, v) \in_G \mathtt{choice} \Leftrightarrow (p, l(v)) \in C \qquad \left(\exists v'.\, (v', v) \in_G \mathtt{nonTerminal}\right) \Leftrightarrow v \in P$$

$$l(v) = \begin{cases} v_1\, l(v_2) & \text{if} \quad (v, v_1) \in_G \mathtt{recogniser} \quad \text{and} \quad (v, v_2) \in_G \mathtt{continuation} \\ \varepsilon & \text{otherwise} \end{cases}$$

To ensure l is well-defined, the labels `recogniser` and `continuation` must describe univalent relations in G ($\mathtt{R}$ is *univalent* iff $(x, y), (x, z) \in_G \mathtt{R}$ implies $y = z$).

Example 3. The following CFG describes the language for CFGs. It omits production rules for non-terminals P and Σ, as Amperspiegel has those production rules built-in. These built-in production rules are the only way to get constant symbols as vertices in the sense of Section 4. We write "↦" for the terminal in Σ, to distinguish it from syntax.

$$\mathtt{S} \mapsto \varepsilon \qquad\qquad \mathtt{S} \mapsto \mathtt{S}\ P\ \texttt{"}\mapsto\texttt{"}\ \mathtt{choice}$$

$$\texttt{nonTerminal} \mapsto P \qquad \texttt{choice} \mapsto \texttt{continuation}$$
$$\texttt{continuation} \mapsto \varepsilon \qquad \texttt{continuation} \mapsto \texttt{recogniser continuation}$$
$$\texttt{recogniser} \mapsto \Sigma \qquad \texttt{recogniser} \mapsto \texttt{nonTerminal}$$

The CFG in Example 3 describes the language of CFGs as used in this paper. It defines a parser yielding parse-graphs with the labels `choice`, `nonTerminal`, `recogniser` and `continuation`. So if G' is the parse graph of some string and the CFG in Example 3, then G' can be interpreted as a CFG in Amperspiegel.

In such G', some vertices are being interpreted as elements of Σ, and some are labels in the parser-graph corresponding to the CFG of G'. These are the vertices that are drawn in black in Figure 2. To ensure G' uses the vertices that were intended, Amperspiegel allows us to write rules to determine equality on vertices in G'. Rules are explained in the next section, but for completeness, we mention the rules necessary with Example 3 for using the graph with the CFG here. They use $\sqsubseteq$ for inclusion, and $\mathbb{1}$ for the identity relation:

$$P \sqsubseteq \mathbb{1} \qquad \Sigma \sqsubseteq \mathbb{1} \qquad \texttt{nonTerminal} \sqsubseteq \mathbb{1}$$

4 Rules

To manipulate graphs in Amperspiegel, the programmer specifies rules. This is done in relation algebra to obtain a declarative, point-free language with attractive algebraic properties. Rules are evaluated with a deduction engine comparable to those for Datalog [3]. To this extent, Amperspiegel maintains a graph containing what it knows, and then makes it more specific by what it can prove. A typical use is to interpret a parse graph as initial knowledge, which is made specific by edges that can be deduced using the rules. This section introduces rules and shows how they are used.

Rules are formed over expressions. Expressions are built from relation symbols $\mathcal{L}$, a reserved symbol $\mathbb{1}$ which stands for the identity relation, and tuples (sets containing exactly one pair) written as $\langle a, b\rangle$ with a and b elements in a set of constants $\mathcal{K}$. We can also use the reserved symbol $\bot$, which stands for the empty relation. These are combined with the operations $_\sqcap_$, $_;_$, and $_^{\smile}$. The operations stand for intersection, relational composition, and relational converse, respectively. For a graph $G = (\mathcal{L}, \mathcal{K} + N, E)$, in which the vertices are *constant symbols* $\mathcal{K}$ or *variable symbols* N, the semantics of an expression $\mathcal{X}$, written as $[\![\mathcal{X}]\!]_G \subseteq (\mathcal{K} + N) \times (\mathcal{K} + N)$, is as in representable relation algebra. We assume $\mathcal{K}$ and N to be disjoint:

$$[\![l]\!]_G = \left\{(x, y) \mid (x, y) \in_G l\right\} \qquad [\![\mathbb{1}]\!]_G = \{(v, v) \mid v \in (\mathcal{K} + N)\}$$
$$[\![\langle a, b\rangle]\!]_G = \{(a, b)\} \qquad [\![\bot]\!]_G = \{\}$$
$$[\![L \sqcap R]\!]_G = [\![L]\!]_G \cap [\![R]\!]_G \qquad [\![L^{\smile}]\!]_G = \left\{(y, x) \mid (x, y) \in [\![L]\!]_G\right\}$$
$$[\![L \,;\, R]\!]_G = \left\{(x, y) \mid \exists z.\ (x, z) \in [\![L]\!]_G \ \wedge\ (z, y) \in [\![R]\!]_G\right\}$$

Definition 5 (Rule). *If L and R are expressions over sets of constant symbols $\mathcal{K}$ and labels $\mathcal{L}$, then $L \sqsubseteq R$ is a* rule. *We say that a graph G* satisfies *a set of rules $\mathcal{R}$, in symbols: $G \vDash \mathcal{R}$, iff for all $(L \sqsubseteq R) \in \mathcal{R}$ we have $[\![L]\!]_G \subseteq [\![R]\!]_G$. We say that a set of rules $\mathcal{R}$* implies *a rule r_0, in symbols: $\mathcal{R} \vDash r_0$, iff for all graphs G we have $(G \vDash \mathcal{R}) \Rightarrow \left(G \vDash \left\{r_0\right\}\right)$.*

4.1 The rule engine by example

We give a flavour of Amperspiegel's deduction engine, by showing how one can reason to construct a non-empty graph that satisfies a set of rules. Consider the example:

Example 4. These rules state that the label l stands for a total and self-inverse relation:

$$\mathbb{1} \sqsubseteq l \,;\, l^{\smile} \tag{1}$$
$$\mathbb{1} \sqsubseteq l \,;\, l \tag{2}$$
$$l \,;\, l \sqsubseteq \mathbb{1} \tag{3}$$

Rule 1 states that l is total, and Rules 2 and 3 say that it is self-inverse.

We construct a non-empty graph G that has no constant symbols, satisfying the rules of Example 4, to illustrate Amperspiegel's rule engine. See Figure 3. Take $G_0 = (\{l\}, \{v_0\}, \{\})$ as initial non-empty graph. We identify a rule that does not hold on G_0, and a pair that shows why it does not. Rule 1 does not hold on G_0 as there must be some v_1 with $(v_0, v_1) \in_G l$. We therefore add the vertex v_1 to G_0, plus an edge from v_0 to v_1 with label l, which gives rise to G_1. On G_1, rule 2 states that some v_2 exists with $(v_0, v_2) \in_G l$ and $(v_2, v_0) \in_G l$. Changing G_1 to fix this adds two more edges and another vertex, giving G_2. Now rule 3 does not hold for $(v_2, v_1) \in [\![l \,;\, l]\!]_{G_2}$. Therefore, we identify v_1 and v_2 giving us G_3. This is a graph for which all rules hold.

4.2 Rule engine semantics

This section explains how Amperspiegel's rule engine is defined. We begin with some notions and notations. We overload a function $f : V_1 \rightarrow V_2$ to a function over sets: $f(V) = \{f(v) | v \in V\}$ for $V \subseteq V_1$, edges: $f(E) = \{(l, f(v_1), f(v_2)) \mid (l, v_1, v_2) \in E\}$, and graphs: $f((\mathcal{L}, V, E)) = (\mathcal{L}, f(V), f(E))$.

Our rule engine gradually changes a graph. We describe these changes in a categorical manner, inspired by Wolfram Kahl [5]. Such a change can be described by a homomorphism, which can be understood as a vertex map that preserves constant symbols and edge labels. This definition is used to describe all graph transformations.

Definition 6 (Graph homomorphism). *Take the graphs with shared sets of labels and constants $G_1 = (\mathcal{L}, \mathcal{K} + N_1, E_1)$ and $G_2 = (\mathcal{L}, \mathcal{K} + N_2, E_2)$. We say that a vertex map $f : \mathcal{K} + N_1 \rightarrow \mathcal{K} + N_2$ is a* graph homomorphism *iff $\forall e \in E_1.\ f(e) \in E_2$, and $\forall k \in \mathcal{K}.\ f(k) = k$.*

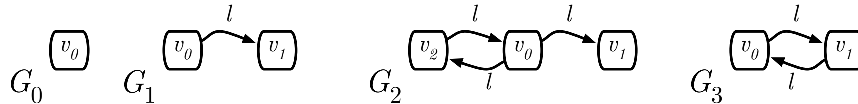

Fig. 3: Applying the rules of Example 4.

If there is a graph homomorphism $f : G_1 \to G_2$, we say that G_2 *is more specific than* G_1, or in symbols: $G_1 \leq G_2$. Graph homomorphisms between graphs with shared sets of labels and constants form a category in which graph homomorphisms are the morphisms. In the following, we assume fixed but arbitrary sets $\mathcal{L}$ of labels and $\mathcal{K}$ of constant symbols.

We use pushouts to combine two graphs. Note that due to the requirement that homomorphisms preserve constants, if $\mathcal{K}$ is non-empty, then the category of graph homomorphism does not have all colimits and not even all pushouts, since constants cannot be identified. For the pushouts that do exist, we introduce an abbreviating notation.

Definition 7 (Pushout along interfaces). *An* interfaced graph *is a pair* (G, s) *where* s *is a sequence of vertices of* G *called interface. Given two interfaced graphs* (G_1, s_1) *and* (G_2, s_2) *with interfaces of the same length* n*, their* pushout along their interfaces*, written* $(G_1, s_1) \sqcup (G_2, s_2)$ *is the interfaced graph* $(G_3, g_1(s_1))$ *where* $G_1 \overset{g_1}{\Longrightarrow} G_3 \overset{g_2}{\Longleftarrow} G_2$ *is the pushout (if existing) of the span* $G_1 \overset{f_1}{\Longleftarrow} G_0 \overset{f_2}{\Longrightarrow} G_2$ *over* $G_0 = (\mathcal{L}, \mathcal{K} + \{x_1, \dots, x_n\}, \{\})$*, and* f_1 *and* f_2 *are graph homomorphisms defined by* $f_i(x_j) = s_i(j)$*.*

We aim to construct the least specific graph G such that $G \vDash \mathcal{R}$, called a least consequence graph. We define this to show correctness of our algorithm.

Definition 8 (Consequence graph). *Given a graph* G_0 *and a set of rules* $\mathcal{R}$ *over the same set of labels and set of constants. We say that* G *is a* consequence graph *of* G_0 *and* $\mathcal{R}$*, if* $G \vDash \mathcal{R}$ *and* $G_0 \leq G$*. Furthermore,* G *is a* least consequence graph *if for each consequence graph* G' *of* G_0 *and* $\mathcal{R}$ *we have* $G \leq G'$*.*

To construct a consequence graph, Amperspiegel repeatedly takes a rule that is not satisfied by a graph, and 'patches' this until there is nothing to repair. For a rule $L \sqsubseteq R$ with a pair in $[\![L]\!]_{G_0}$ that is not in $[\![R]\!]_{G_0}$, we do a *step*: A *patch* is created with the shape of R, which is combined into G_0 with a pushout.

Definition 9 (Patch). *The* patch *of an expression* $\mathcal{X}$ *over sets of labels* $\mathcal{L}$ *and constants* $\mathcal{K}$*, in symbols* $\big(G, (v_1, v_2)\big) = \Delta(\mathcal{X})$*, is a graph over* $\mathcal{L}$ *and a pair of vertices in that graph, inductively defined:*

$$
\begin{aligned}
\Delta(\mathbb{1}) &= ((\mathcal{L}, \mathcal{K} + \{1\}, \{\}), (1, 1)) \\
\Delta(R \sqcap S) &= \Delta(R) \sqcup \Delta(S) \\
\Delta(R \mathbin{;} S) &= (G', (v_1, v_4)) \\
&\qquad \textit{where } \ (G', _) = (G_R, (v_2)) \sqcup (G_S, (v_3)) \\
&\qquad \textit{and } \ (G_R, (v_1, v_2)) = \Delta(R) \ \textit{ and } \ (G_S, (v_3, v_4)) = \Delta(S) \\
\Delta(R^{\smile}) &= (G', (v_2, v_1)) \quad \textit{where} \quad (G', (v_1, v_2)) = \Delta(R) \\
\Delta(\langle a, b \rangle) &= ((\mathcal{L}, \mathcal{K}, \{\}), (a, b)) \\
\Delta(l) &= \big((\mathcal{L}, \mathcal{K} + \{v_1, v_2\}, \{(l, v_1, v_2)\}), (v_1, v_2)\big)
\end{aligned}
$$

As with pushouts, $\Delta(\mathcal{X})$ *may not be defined. Also,* $\Delta(\bot)$ *is intentionally left undefined.*

Another example for an expression with undefined patch is $\mathbb{1} \sqcap \langle a, b\rangle$, if $a \neq b$, since the necessary pushout would have to identify the constant symbols a and b.

We use patches to work towards a consequence graph. This is done stepwise through $\mathcal{R}$-steps, that are given by the set of rules.

Definition 10 **($\mathcal{R}$-step).** *Let G be a graph. Let $(L \sqsubseteq R)$ be a rule, and let p be a pair of vertices in G such that:*

$$p \in [\![L]\!]_G \qquad\qquad p \notin [\![R]\!]_G$$

Then $G \xrightarrow[p]{L\sqsubseteq R} G'$ is a step *where $G' = \Delta(R) \sqcup (G, p)$ if defined, and $G' = \lightning$ otherwise. If $\mathcal{R}$ is a set of rules, then $G \xrightarrow{\mathcal{R}} G'$ is an $\mathcal{R}$-*step *if there exists a rule $r \in \mathcal{R}$ and a pair of vertices p in G such that $G \xrightarrow[p]{L\sqsubseteq R} G'$. If there is no $\mathcal{R}$-step for a graph G, then we say G is in $\mathcal{R}$-normal form. For notational convenience, $\xrightarrow{\mathcal{R}}$ is an endo-relation on the disjoint union of $\lightning$ with graphs, where $\lightning$ counts as an additional $\mathcal{R}$-normal form.*

Correctness of '$\mathcal{R}$-step' is understood as follows: If there is a terminating sequence $G_0 \xrightarrow{\mathcal{R}} \cdots \xrightarrow{\mathcal{R}} G_n \neq \lightning$, then G_n is a *least consequence graph* of G_0. This follows from observing that if $G_i \xrightarrow{\mathcal{R}} G_{i+1}$, then $G_i \leq G_{i+1}$. If G is a consequence graph of G_i and $\mathcal{R}$, then G is also a consequence graph of G_{i+1} and $\mathcal{R}$. This holds in particular if G is a least consequence graph. Finally, if G_n is a graph in $\mathcal{R}$-normal form, then G_n is a least consequence graph of G_n and $\mathcal{R}$. Furthermore, if $G \xrightarrow{\mathcal{R}} \lightning$, then there is no consequence graph of G and $\mathcal{R}$. This shows soundness of finding a consequence graph through a normalising sequence $G_0 \xrightarrow{\mathcal{R}} G_1 \cdots \xrightarrow{\mathcal{R}} G_n$ in which G_n is either $\lightning$ or a least consequence graph, which is what Amperspiegel's rule engine does.

Note that $\xrightarrow{\mathcal{R}}$ need not be weakly normalising or confluent, and the order in which we apply rules can determine whether we reach a normal form. It is possible to have an infinite sequence of $\mathcal{R}$-steps even though there are terminating sequences. To make this less likely, Amperspiegel ensures fairness: A sequence $G_0 \xrightarrow{\mathcal{R}} G_1 \cdots$ is fair if for all pairs p there are finitely many i such that $G_i \xrightarrow[p]{r} _$. This condition is implemented by imposing a total order on the vertices, treating smallest vertices first, and making new vertices the largest elements in this order.

Amperspiegel's rule engine can terminate by finding the least consequence graph, or discovering that no such graph exists by reaching $\lightning$. The possibility of non-termination makes it that it is not a decision procedure. We leave the question whether Amperspiegel implements a semi-decision procedure as future work. We conjecture that the problem whether no least consequence graph exists is undecidable, yet semi-decidable, and that our procedure is a semi-decision procedure.

5 Amperspiegel's embedding of the rule engine

This section shows how Amperspiegel uses the rule engine of the previous section to implement more general graph transformations, including destructive rules. We apply a

rule system using the `apply` switch, which gets three arguments: a graph that encodes the rules $\mathcal{R}$, the name of a source graph $G_s = (\mathcal{L}_s, V_s, E_s)$, and the name for a target graph $G_t = (\mathcal{L}_t, V_t, E_t)$. The label set for rules is $\mathcal{L}_s + \mathcal{L}' + \mathcal{L}_t$. To ensure disjointness of these three sets of labels, `pre`, `during` and `post` are used as a prefix to labels respectively. The graph of which a least consequence graph is calculated is $G_0 = (\mathcal{L}_s + \mathcal{L}' + \mathcal{L}_t, V_s, E_s')$, in which E_s' contains the appropriately relabelled edges of E_s. The least consequence graph of G_0 and the rules $\mathcal{R}$ is then $G = (\mathcal{L}_s + \mathcal{L}' + \mathcal{L}_t, V_t, E)$. The target graph has the edges $E_t = \{(r, x, y) \mid (\mathsf{post}\ r, x, y) \in E\}$, where post is the rightmost constructor of the disjoint union $\mathcal{L}_s + \mathcal{L}' + \mathcal{L}_t$.

Consequently, the graph the procedure starts with only contains edges of G_s. The target graph will be overwritten. After obtaining the consequences by running the procedure, we only look at the edges that are in $\mathsf{post}(r)$ for some r and put those in G_t. For convenience, we allow labels of the form $\mathsf{during}(r)$, to allow labels for edges that do not end up in G_t, but are also guaranteed not to be used in G_s.

The user can use her own rules in Amperspiegel, as the rules are described as a graph. This follows the same pattern as describing a CFG with a graph. For an expression e, there is a pair (e, p) with p uniquely determined by e:

$$(e, p) \in_G \mathtt{conjunct} \cup \mathtt{compose} \cup \mathtt{converse} \cup \mathtt{pair} \cup \mathtt{pre} \cup \mathtt{during} \cup \mathtt{post} \cup \mathtt{id}$$

such that (e, p) occurs in exactly one of the relations mentioned, say l. If l is `conjunct` or `compose`, there are unique e_1 and e_2 such that $(p, e_1) \in_G$ `eFst` and $(p, e_2) \in_G$ `eSnd`. These e_1 and e_2 are, in turn, expressions again. If l is `pre`, `during` or `post`, p is a relation name (an unquoted string in $\mathcal{K}$). For `converse`, p is an expression. For `pair`, p is a pair of strings (quoted or unquoted) that can be accessed through the relations `pFst` and `pSnd`. If $l =$ `id`, p does not matter. A set of rules is a relation between expressions.

To take full advantage of rules as graphs, Amperspiegel allows a graph to contain both a grammar and rules, given by taking the union of the corresponding triples. We use these two together, by a switch called `-Parse` (note the capital `P`), that first parses and then applies the rules to the result. This makes many syntactical extensions straightforward to achieve. Take for instance the operation $\mathtt{dom}(R)$, containing all pairs (x, x) for which x is in the domain of R, defined as follows:

$$\mathtt{dom}(R) = (R\,;(R^{\smile})) \sqcap \mathbb{1}$$

We allow the relation `dom` to be used without changing Amperspiegel, by adding the following rule to the parser (for readability, we underline labels instead of writing `post`):

$$\mathtt{pre\ dom} \sqsubseteq \underline{\mathtt{conjunct}}\,;(\underline{\mathtt{eFst}}\,;\underline{\mathtt{compose}}\,;(\underline{\mathtt{eFst}} \sqcap \underline{\mathtt{eSnd}}\,;\underline{\mathtt{converse}}) \sqcap \underline{\mathtt{eSnd}}\,;\underline{\mathtt{id}})$$

With this, we have seen an example of using rules in order to extend the syntax of rules. Section 7 contains another example where a syntax extention was useful.

6 Printing

We consider printer as a reverse operation to parsing. It is not always possible to reconstruct the original string. Consider for instance the following CFG, for lists with at least

two words:

$$\texttt{Start} \mapsto \texttt{Word Word} \qquad \texttt{Start} \mapsto \texttt{Word Start}$$
$$\texttt{Word} \mapsto \texttt{e a t} \qquad \texttt{Word} \mapsto \texttt{t e a}$$

Printing of graphs that contain only univalent relations can be done unambiguously if for every non-terminal, each symbol occurs at most once on the right hand side of its production rules. We change the CFG to meet this condition, without changing the language it accepts:

$$\texttt{Start} \mapsto \texttt{Word}_1\ \texttt{Word}_2 \qquad \texttt{Start} \mapsto \texttt{Word Start} \qquad \texttt{e}' \mapsto \texttt{e}$$
$$\texttt{Word} \mapsto \texttt{e}'\ \texttt{a}'\ \texttt{t}' \qquad \texttt{Word} \mapsto \texttt{t e a} \qquad \texttt{a}' \mapsto \texttt{a}$$
$$\texttt{Word}_1 \mapsto \texttt{Word} \qquad \texttt{Word}_2 \mapsto \texttt{Word} \qquad \texttt{t}' \mapsto \texttt{t}$$

When printing graphs that aren't a parse graph, we may encounter relations that are not univalent. For this purpose, we add the label `separator` to a graph describing a CFG, in addition to the four existing labels. The type of edges with this label can be thought of informally as $(P + \Sigma) \times \Sigma$, although Amperspiegel does not consider any structure on vertices.

The syntax for a printer closely follows that of a parser. The main difference is that we allow a relation to be named between square brackets, along with an optional separator string. This means that we can largely reuse the parser for a CFG as defined earlier. We drop the production-rule `recogniser` $\mapsto$ `nonTerminal` from Example 3, and replace it with:

$$\texttt{recogniser} \mapsto \texttt{idNonTerminal}$$
$$\texttt{recogniser} \mapsto \texttt{"["\ recRelation\ "]"\ nonTerminal}$$
$$\texttt{recogniser} \mapsto \texttt{"["\ recRelation\ "SEPBY"\ separator\ "]"\ nonTerminal}$$

One can think of `idNonTerminal` as a typed identity relation for those instances where we want to use the nonTerminal symbol as a label.

We recognise `recRelation` and `separator` as strings, and use the following rules:

$$\texttt{idNonTerminal} \sqsubseteq \mathbb{1} \qquad \texttt{recRelation} \sqsubseteq \mathbb{1}$$
$$\texttt{idNonTerminal} \sqsubseteq \texttt{nonTerminal}$$

7 Using Amperspiegel to transform ArchiMate files into Ampersand code

In previous sections we discussed parsing, rules to evaluate, and printing. These are the necessary ingredients for transforming data structures. To demonstrate that Amperspiegel can do nontrivial work, it has been put to the test of practice. We picked a problem that was being solved at the time of writing in a software project in the Dutch government: to transform source code from ArchiMate [8] to Ampersand [4].

The specifics of the tools Ampersand and ArchiMate are not important to understand the transformation, but we give a little background: ArchiMate is a modeling tool to get an overview of a business, similar to UML yet more coarse grained. The tool helps users to build, visualise and modify architectures cooperatively, but does not feature a way to turn such architectures into code. For this purpose, we are interested in using another tool that describes architectures that does produce code, namely Ampersand. Ampersand can generate web-applications based on architectures, but often an architecture is already described in another language, in our case: ArchiMate.

To understand the transformation, it suffices to know that ArchiMate files are XML files describing 'elements'. Between these elements there are 'relations'. Elements are things like actors, business components, services, and infrastructure. A relation can be 'implements', describing which infrastructures implement which services.

The purpose of this section is to describe how one can create transformations with Amperspiegel. We define an XML parser, interpret the resulting graph as an ArchiMate model, and turn it into an Ampersand model. This section uses verbatim Amperspiegel syntax.

In the development of the XML parser, we keep the specification of syntax and rules in a single file. This changes the syntax for describing a CFG slightly: Each line should end with a dot, to keep the grammar unambiguous. We form rules, using `|-` as notation for $\sqsubseteq$, prefixed with `RULE`. We use `KEEP relationName` as syntax-sugar for:

```
RULE pre relationName |- post relationName
```

To achieve this, the parser for CFG's populates the relation `keep`, and the set of rules that is then applied to the result contains the rule:

```
RULE pre keep |- post rule;(post eFst;post pre /\ post eSnd;post post)
```

Similarly, `[expression -> elementName]` is a shorthand for the expression:

```
expression;<elementName,elementName>;expression~ /\ I
```

These short-hands are useful for the development of the XML parser and the transformation that follows it. We used them without changing Amperspiegel itself. We changed the Amperspiegel-scripts that define the parser for Amperspiegel-scripts instead. In the parser, `"[" pointExpression "->" pointElement "]"` is added in the right hand side of a production-rule for an expression. We also add these rules:

```
RULE pre pointExpression |- (post conjunct;(post eFst;(post compose;(post
         eFst /\ ((post eSnd;post compose);(post eSnd;post converse))))))
RULE pre pointElement |- (post conjunct;((post eFst;((((post compose;
        post eSnd);post compose);post eFst);pre pair)) /\ post eSnd))
```

7.1 Parsing XML

Building an XML parser lies outside of the scope what Amperspiegel was initially intended for: parsing Ampersand-like scripts. Consequently, Amperspiegel's lexer is not designed for parsing XML; it ignores comments and whitespace. Fortunately, we can get away with this by restricting ourselves to XML without text. This means tags, including

attributes, are fine, but `<tag>text like this</tag>` is not. Such a tag would have to be replaced by an attribute-value, such as: `<tag value="text like this" />`.

An XML parser can then be defined as follows (`Start` is Amperspiegel's start symbol for a CFG):

```
Start > "<?xml" attributeList "?>" tagList.
Start > tagList.
tagList > tag tagList.
tagList > .
tag > "<" tagName attributeList ">" tagList "</" tagName ">".
tag > "<" tagName attributeList "/>".
tagName > UnquotedString .
attributeList > attribute attributeList.
attributeList > .
attribute > attributeName "=" attributeValue.
attributeValue > QuotedString.
attributeName > UnquotedString.

RULE  pre UnquotedString |- I
RULE  pre QuotedString |- I
RULE  pre tagList |- I
RULE  pre attributeList |- I
RULE  (pre tagName) ~ ; pre tagName |- I     -- univalence of tagName

KEEP attributeName   KEEP attributeValue   KEEP attribute
KEEP tagName         KEEP tag
```

The first lines describe a CFG for XML. Note that the lines end with a dot, in order to distinguish `KEEP` statements from a continuation in which `KEEP` acts as recogniser. The rules for `tagList` and `attributeList` cause `tag` and `attribute` to be relations, rather than partial functions from the head of the list. We can forget the order-information of `attributeList` and `tagList` since for ArchiMate this order is irrelevant.

The rule for univalence of `tagName` requires a closing tag to match the opening tag, because the parser generates two `tagName` edges from the first `tag` rule to different tag names, which, after the contraction of `UnquotedString` edges, are string constant symbols. Parsing `<openingtag></closingtag>` will result in trying to identify two constants in $\mathcal{K}$ and produce the message:

```
Rules caused "openingtag" to be equal to "closingtag"
```

The XML we parse is well-formed, so these errors do not occur in practice.

7.2 Transforming a graph

We parse XML such as the following. Figure 4 shows the first two lines parsed:

```
<element identifier="id-1311" xsi:type="BusinessProcess">
  <label xml:lang="en" value="Collect Premium"/></element>
<element identifier="id-1208" xsi:type="BusinessService">
  <label xml:lang="en" value="Premium Payment Service"/></element>
```

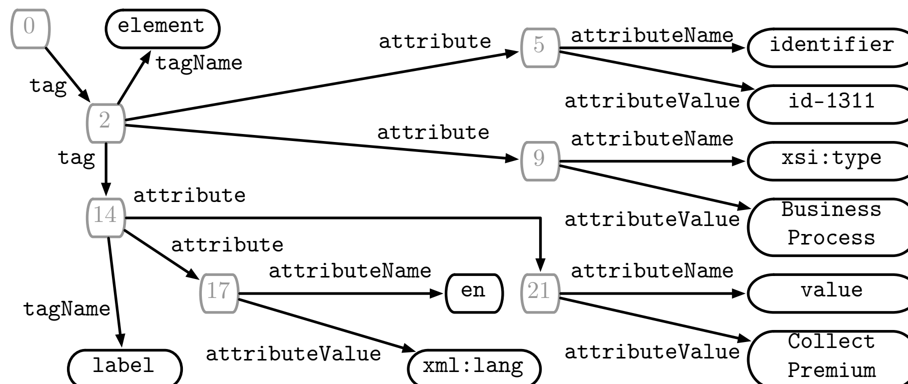

Fig. 4: Graph from applying parser and rules of Section 7.1 to two lines of XML.

```
<relationship identifier="id-1329" source="id-1311"
  target="id-1208" xsi:type="RealisationRelationship" />
```

The corresponding Ampersand code we will transform this XML into is:

```
CLASSIFY BusinessProcess ISA Element
CLASSIFY BusinessService ISA Element
RELATION RealisationRelationship :: Element * Element
POPULATION [ ( "Collect Premium" , "Premium Payment Service" ) ]
```

Here are some of the rules which we use to transform the parsed XML:

```
RULE pre attribute;[pre attributeName -> identifier]
     ; pre attributeValue |- I
RULE pre tag;[pre tagName -> label] |- during lab
RULE pre attribute; [pre attributeName -> value]
     ; pre attributeValue |- during value
RULE during lab; during value |- post label
```

The first rule states that identifiers are unique to elements, allowing us to use these as handlers. The second introduces a temporary abbreviation `lab` for `<label>` tags. The third introduces the abbreviation `value` for `value` attributes. The last creates the relation `label` from the value of pairs in `lab`.

To obtain all element types without duplicates, we use these rules:

```
RULE pre attribute; [pre attributeName -> xsi:type]
     ; pre attributeValue |- during dtype
RULE pre tag; [pre tagName -> element] |- during element
RULE during element; during dtype |- during X ; post type
RULE post type |- I
```

The first two rules create temporary shorthands: `dtype` and `element`. The third rule looks only at the element types, and creates a tuple in `type` with that target (and a fresh source). The fourth rule states that the source of that tuple should be equal to the target, removing duplicates. Finally, we obtain all relations and their triples:

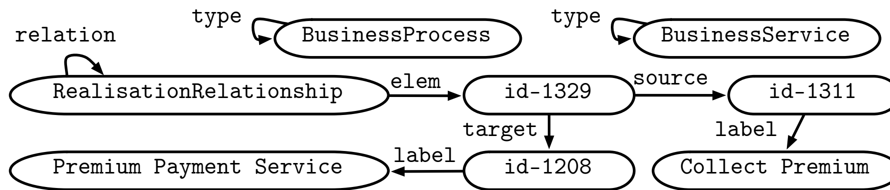

Fig. 5: The triples after applying the rules of Section 7.2

```
RULE [pre tagName -> relationship];during dtype |- post elem~
RULE pre attribute; [pre attributeName -> source] ; pre attributeValue
     |- post source
RULE pre attribute; [pre attributeName -> target] ; pre attributeValue
     |- post target
RULE post elem ; post elem ~ /\ I |- post relation
```

Figure 5 shows the triples computed by Amperspiegel for the XML excerpt.

7.3 Printing a graph

We define a printer such that there are no `identifier` values in the final output. Since the relations are not necessarily well typed in ArchiMate files, we create a type 'Element' to stand in for any type.

The printer is defined as follows:

```
Start > [I SEPBY "\n"] Statement.
Statement > "CLASSIFY" [type] UnquotedString "ISA Element".
Statement > "RELATION" [relation] UnquotedString
      ":: Element * Element\nPOPULATION [" [elem SEPBY "\n , "] Pair "]".
Pair > "(" [source] Labeled "," [target] Labeled ")".
Labeled > [label] String.
```

The relation `I` is used in the first line of the printer. This determines which statements to print, and which not. For our purpose, we print all statements, by adding the rules:

```
RULE post type |- post I
RULE post relation |- post I
```

To summarise how we use Amperspiegel's tool-chain:

- Parse a CFG describing an XML parser in the file `xml.cfg`. To the result, apply the rules for CFGs. Put the result in the graph '`xml`'. On the commandline of Amperspiegel we write: `-Parse xml.cfg cfg xml`.
- Parse rules to convert the XML data specific to ArchiMate, and the corresponding printer specific to Ampersand. The corresponding file is `archi.cfg`. To Amperspiegel we pass: `-Parse archi.cfg cfg archi`
- Parse the ArchiMate xml file `Archisurance.xml` and apply the rules that go with the XML parser. This uses the graph '`xml`': `-Parse Archisurance.xml xml`. Since we omit the third argument, the result is put in the graph '`population`'.

– Apply the rules in the graph 'archi' to population. Put the result in population: -apply archi.
– Print the graph 'population' using the printer defined in 'archi'. In Amperspiegel: -print archi.

We sequence the listed operations on the command line:

```
Amperspiegel -Parse xml.cfg cfg xml -Parse archi.cfg cfg archi \
          -Parse Archisurance.xml xml -apply archi -print archi
```

For the example XML code of Section 7.2, this produces exactly the mentioned Ampersand code. Parsing and printing a file of about 600 lines produces 209 lines in eleven seconds.

8 Discussion

Most parser implementations are a partial function from strings to finite tree structures. We use a standard parsing algorithm, and turn the result into a graph. Consequently, CFGs that generate infinite trees yet finite graphs remain future work.

Applying rules is slow: Amperspiegel traverses the right hand side expressions for every pair and applies the patch as it constructs it. Sharing work between applications of a rule may improve performance. We plan to use Amperspiegel to generate code out of a set of rules, hopefully boosting the performance of Amperspiegel. Ideally, we would also use Amperspiegel to generate code out of a CFG or a printer, making the core of Amperspiegel even simpler. As mentioned, Amperspiegel only consists of a thousand lines of Haskell code. We hope to further reduce this number in the process.

Conclusion We introduced Amperspiegel, and used it for a source-to-source transformation, producing Ampersand code from ArchiMate code. To do so, the Amperspiegel syntax was extended in a convenient manner. This shows how triple graphs can be used to describe simple programs in a flexible, modular way.

Acknowledgements I thank Wolfram Kahl for helping me greatly improve this paper's clarity in an intensive process of iterative feedback. I also thank the anonymous reviewers and Stef Joosten for their comments on an earlier version of this paper. Supported by the Austrian Science Fund (FWF) project Y757.

References

1. van den Brand, M., Visser, E.: Generation of formatters for context-free languages. ACM Transactions on Software Engineering and Methodology (TOSEM) 5(1), 1–41 (1996)
2. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. a language and toolset for program transformation. Science of computer programming 72(1), 52–70 (2008)
3. Gottlob, G., Lukasiewicz, T., Pieris, A.: Datalog+/-: Questions and answers. In: Proceedings of the Fourteenth International Conference on Principles of Knowledge Representation and Reasoning (KR). pp. 682–685 (2014)

4. Joosten, S.: Software development in relation algebra with Ampersand. In: Pous, D., Struth, G., Höfner, P. (eds.) Relational and Algebraic Methods in Computer Science: 16th International Conference (RAMICS). Springer International Publishing (2017)
5. Kahl, W.: Algebraic graph derivations for graphical calculi. In: International Workshop on Graph-Theoretic Concepts in Computer Science. pp. 224–238. Springer (1996)
6. Kats, L.C., Visser, E.: The Spoofax language workbench: Rules for declarative specification of languages and ides. In: ACM Sigplan Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'10). vol. 45, pp. 444–463. ACM (2010)
7. Klint, P., van der Storm, T., Vinju, J.: RASCAL: A domain specific language for source code analysis and manipulation. In: Proceedings of the 2009 9th IEEE International Working Conference on Source Code Analysis and Manipulation. pp. 168–177. SCAM '09, IEEE Computer Society, Washington, DC, USA (2009), http://dx.doi.org/10.1109/SCAM.2009.28
8. Lankhorst, M.M., Proper, H.A., Jonkers, H.: The architecture of the ArchiMate language. In: Enterprise, business-process and information systems modeling, pp. 367–380. Springer (2009)
9. Robertson, E.L.: Triadic relations: An algebra for the semantic web. In: Proceedings of the Second International Conference on Semantic Web and Databases. pp. 91–108. SWDB'04, Springer-Verlag (2005), http://dx.doi.org/10.1007/978-3-540-31839-2_8