Proceedings of the Sixth International Workshop on Graph Computation Models

July 20, 2015. L'Aquila, Italy

GCM 2015

Graph Computation Models

Detlef Plump (editor)

Preface

This volume contains the proceedings of the Sixth International Workshop on Graph Computation Models (GCM 2015). The workshop took place in L'Aquila, Italy, on 20 July 2015, as part of STAF 2015 (Software Technologies: Applications and Foundations) and co-located with ICGT 2015 (International Conference on Graph Transformation). The aim of the GCM workshop series is to bring together researchers interested in all aspects of computation models based on graphs and graph transformation. It promotes the cross-fertilizing exchange of ideas and experiences among researchers and students from the different communities interested in the foundations, applications, and implementations of graph computation models.

Previous editions of GCM were held in Natal (Brazil) in 2006, in Leicester (UK) in 2008, in Enschede (The Netherlands) in 2010, in Bremen (Germany) in 2012, and in York (UK) in 2014.

These proceedings contain the abstract of an invited talk and seven accepted papers. Each submission was carefully reviewed by three Programme Committee members. The topics of the papers include applications of graph transformation to parsing and policy modelling, parallel execution models of graph transformation, the theoretical foundations of graph transformation with attributes, and the verification of graph programs. Revised selected papers from these proceedings will be published as an issue of the international journal Electronic Communications of the EASST.

I would like to thank all who contributed to the success of GCM 2015, especially the authors and the Programme Committee for its work in the selection process. I would also like to thank the ICGT 2015 Programme Chairs and the organizers of STAF 2015 for their support in organizing GCM 2015.

July 20, 2015
L'Aquila, Italy

Detlef Plump (editor)

Organisation

Chair

Detlef Plump	The University of York, UK

Program Committee

Rachid Echahed	Laboratoire d'Informatique de Grenoble, France
Maribel Fernandez	King's College London, United Kingdom
Annegret Habel	Universität Oldenburg, Germany
Dirk Janssens	Universiteit Antwerpen, Belgium
Barbara König	Universität Duisburg-Essen, Germany
Hans-Jörg Kreowski	Universität Bremen, Germany
Mohamed Mosbah	LaBRI, Université de Bordeaux, France
Detlef Plump	The University of York, United Kingdom

Additional Reviewers

Nils Erik Flick	Universität Oldenburg, Germany
Ivaylo Hristakiev	The University of York, United Kingdom
Dennis Nolte	Universität Duisburg-Essen, Germany
Christoph Peuser	Universität Oldenburg, Germany
Christopher Poskitt	ETH Zürich, Switzerland
Hendrik Radke	Universität Oldenburg, Germany

Table of Contents

20 Years of Concurrent Model Driven Application Engineering with Triple Graph Grammars (Abstract)

Andy Schürr

FG Real-Time Systems
Technische Universität Darmstadt
Germany

Today model-driven system development is a well-established and successfully used paradigm in many engineering disciplines including e.g. mechanical, electrical, automation, and software engineering. Models are, therefore, used and concurrently manipulated in different disciplines at different levels of abstractions across the whole lifecycle of a single product. Keeping all these models and related other engineering artefacts synchronized often turns out to be a nightmare. The situation even becomes worse when product line engineering principles are used to develop a family of products instead of a single product on top of a common platform.

Model transformation techniques promise to be a silver bullet for the construction and validation of reliable and efficiently working model synchronizers on a new level of abstraction. Triple Graph Grammars (TGGs) are a bidirectional model transformation formalism, where a single specification generates a language of related graph tuples (pairs of models) together with an intermediate correspondence graph (traceability link database). A single TGG specification is used as input for a compiler that generates batch transformations as well as incrementally working synchronizers that assist engineers in their daily work to keep networks of evolving models and their traceability relationships in a consistent state.

This talk starts with an introduction to concurrent model driven application engineering in general, explains how triple graph grammars are specified and used to generate model transformation and integration tools, and finishes with an assessment of the state-of-the-art of 20 years of TGG related research activities.

Coupled Transformations of Shared Packed Parse Forests

Vadim Zaytsev

Universiteit van Amsterdam, The Netherlands, vadim@grammarware.net

Abstract. SPPF (shared packed parse forest) is the best known graph representation of a parse forest (family of related parse trees) used in parsing with ambiguous/conjunctive grammars. Systematic general purpose transformations of SPPFs have never been investigated and are considered to be an open problem in software language engineering. In this paper, we motivate the necessity of having a transformation operator suite for SPPFs and extend the state of the art grammar transformation operator suite to metamodel/model (grammar/graph) cotransformations.

1 Motivation

Classically, parsing consumes a string of characters or tokens, recognises its grammatical structure and produces a corresponding parse tree [1,52]. However, sometimes we end up in situations when trees are not expressive enough. The most common scenarios include generalised parsing and Boolean grammar-based parsing. *Generalised parsing algorithms* (GLR [43], SGLR [44], GLL [39], RIGLR [38], etc) differ from the classic ones in dealing with ambiguities [7]: instead of trying to avoid, ignore or report them, ambiguous parses result in so called parse forests — sets of equally grammatically correct parse trees. In practice, these sets usually need to be filtered or ranked in order to make full use of the available tree-based approaches to program analysis and transformation. In *Boolean grammars* [34] and *conjunctive grammars* [33], we have conjunctive clauses in a grammar as first class citizens and must treat them properly when parsing, which means having special kinds of nodes in a parse tree whose descendant subtrees share leaves [35]. Both kinds of structures defined by these two related approaches conceptually are *parse forests*.

There have been various attempts in the past to represent parse forests. The earliest ones required a grammar to be in a Chomsky Normal Form [11] — theoretically a reasonable assumption since any context-free grammar can be normalised to CNF, but ultimately we need a parse forest for the original grammar, not for the normalised one, which would require bidirectional grammar transformations [46] to be coupled with tree and forest transformations, which is far from trivial.

The next attempt in representing parse forests revolved around tree merging [14]: such a parse forest representation would result in a tree-like DAG with

all the edges of all the trees in the forest. This is obviously an overapproximation of the forest (see Figure 1), which requires additional information in order to be unfolded into a set of trees — in other words, in order for any sensible manipulation to happen. Obviously, having a data structure that requires so much nontrivial postprocessing overhead, is highly undesirable.

The best representation of a conceptual parse forest (a set of trees with equal lists of leaves) so far is a so-called *shared packed parse forest* [43, §2.4], SPPF from now on: its components are merged from the top until the divergent nodes, and due to maximal sharing the leaves and perhaps even entire subtrees grouping leaves together, are also merged. An example of such a graph is given on Figure 2. Formally, an SPPF is an acyclic ordered directed graph where each edge is a tuple from a vertex to a linearly ordered list of successors and each vertex may have more than one successor list. If V is a set of vertices, then edges are:

$$E = \{\langle v_i, (v_{i1}, v_{i2}, ..., v_{ik_i})\rangle \mid v_i \in V, v_{ij} \in V\} \subseteq V \times V^*$$

SPPF-like structures are used nowadays both in software language toolkits that allow explicit ambiguities (such as Rascal [22]) and those that allow explicit conjunctive clauses (such as TXL [42]). For a detailed view on the implementation details we refer the readers to a paper on ATerms [5]. However, the theory of their transformations is underdeveloped — this was pointed out as one of the major open problems in modern software language engineering by James Cordy and explained in his recent keynote at the OOPSLE workshop [3].

2 Transformation

For many years trees have been the dominant data structure for representing hierarchical data in software language processing. They are remarkably easy to define, formalise, implement, validate, visualise and transform. There are many ways to circumvent data representation as graphs by considering a tree together with a complementary component such as a relation between its vertices that would have turned a tree into a cyclic graph, as well as many optimisations of graph algorithms that work on skeleton trees of a graph. Take, for instance, traversing a tree — it can be done hierarchically from the root towards the leaves or incrementally from the leaves towards the root, each case guaranteed termination even if the traversal is not supposed to stop when a match is made. This naturally provides us with four traversal strategies found in metaprogramming: bottom-up-continue, bottom-up-break, top-down-continue and top-down-break [8,22]. More sophisticated and flexible traversal strategies exist (e.g., Nuthatch [2]), but the actual need for them is rather rare. For a detailed overview of visiting functions, strategic programming and typed/untyped rewriting we refer the readers to the work of van den Brand et al [9] and the bibliography thereof. This section is focused on finding existing techniques that can be or are in fact SPPF transformations.

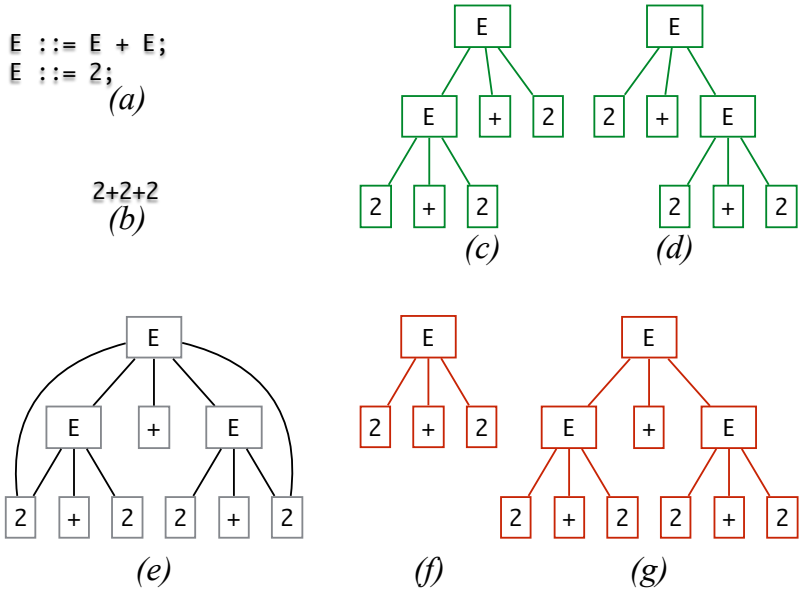
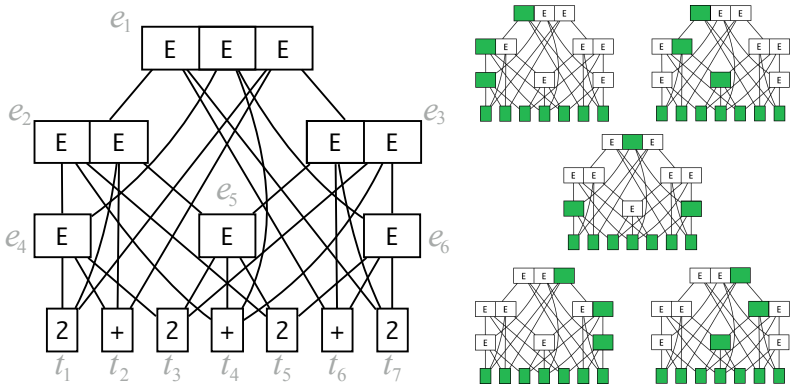


Fig. 1. Demonstration that the Earley representation overapproximates parse forests: **(a)** a simple ambiguous grammar example; **(b)** a term with ambiguous parse; **(c)**&**(d)** correct parses; **(e)** the graph representation of the forest suggested by Earley [14]; **(f)**&**(g)** incorrect parse trees that are well-formed according to the grammar (a) and covered by the parse tree representation (e), but not corresponding to the actual term (b).

2.1 Disambiguation

One of the relatively well-researched kind of SPPF transformations is disambiguation — it is commonly practised with ambiguous generalised parsing because static detection of ambiguity is undecidable for context-free grammars [10]. However, most of the time the intention of an average grammarware engineer is to produce one parse tree, so this line of research is mostly about leveraging additional sources of information to obtain a parse tree from a parse forest. There are three main classes of disambiguation techniques:

- ◇ Ordered choice, dynamic lookahead and other conventions aimed to *prevent* ambiguities altogether or avoid them. These are fairly static, relatively well-understood and widely used in TXL [12], ANTLR [37] and PEG [16].
- ◇ Follow/precede restrictions, production rule priorities, associativity rules and other annotations for local sorting (preference, avoidance, priorities) that help to prune the parse forest *during* its creation. Since these are algorithmic approaches in a sense that they modify the generation process of an SPPF and thus are not proper mappings from SPPFs to SPPFs, we will not consider



$$\{\ \langle e_1,(e_2,t_6,t_7)\rangle, \langle e_1,(e_4,t_4,e_6)\rangle, \langle e_1,(t_1,t_2,e_3)\rangle,$$
$$\langle e_2,(e_4,t_4,t_5)\rangle, \langle e_2,(t_1,t_2,e_5)\rangle, \langle e_3,(e_5,t_6,t_7)\rangle, \langle e_3,(t_3,t_4,e_6)\rangle,$$
$$\langle e_4,(t_1,t_2,t_3)\rangle, \langle e_5,(t_3,t_4,t_5)\rangle, \langle e_6,(t_5,t_6,t_7)\rangle\ \}$$

Fig. 2. One the left, an SPPF graph resulted from parsing the input "2+2+2+2" with the grammar from Figure 1 (a). On the right, there are five parse trees in a forest, which are packed in a triple ambiguity, two of subgraphs of which have double ambiguities. All of them share leaves and subtrees whenever possible. Below the pictures we show its formal representation as an ordered directed graph.

them in the rest of the paper and refer to other sources primarily dedicated to them [7,4].

- ◇ Disambiguation filters that are run *after* the parsing process has yielded a fully formed SPPF: their main objective is to reduce the number of ambiguities and ultimately to shave all of them off, leaving one parse tree. An example of this would be how processing production rules marked for rejection is done for SGLR [7] and GLL [4] — even though recursive descent parsers can handle an equivalent construct (and-not clause) during parsing without any trouble [42].

Formally speaking, the first class never produces parse forests; the second class works with disambiguators (higher order functions that take a parser and return a parser that produces less ambiguous SPPFs) [7]; the third class uses filters (functions that take an SPPF and produce a less ambiguous SPPF) [23]. In some sources approaches with disambiguators are called "semantics-directed parsing" and approaches with filters are called "semantics-driven disambiguation" [6], since both indeed rely on semantic information to aid in the syntactic analysis. Disambiguation filters are still but a narrow case of SPPF transformation, but they have apparent practical application and are therefore well-researched.

2.2 Grammar programming

Grammar programming is like normal programming, but with grammars: there is a concrete problem at hand which can be solved with a grammar, which is then being adjusted until an acceptable solution emerges. A representative pattern here is working with a high level software artefact describing a language (we assume it to be a grammar for the sake of simplicity, but in a broad sense it can be a schema, a metamodel, an ontology, etc), from which a tool solving the problem at hand is inferred automatically.

There are at least three common approaches to grammar programming: manual, semi-automated and operator-based. *Manual grammar programming* involves textual/visual editing of the grammar file by a grammarware engineer. It is the easiest method in practice and is used quite often, especially for minor tweaks during grammar debugging. However, it leads to hidden inconsistencies within grammars (which require advanced methods like grammar convergence to uncover [31]), between changed grammars and cached trees (which demand reparsing) and between grammars and program transformations (which requires more manual labour). *Semi-automated grammar programming* adds a level of automation to that and thus is typically used in scenarios when a baseline grammar needs to be adjusted in different ways to several tasks (parsing language dialects, performing transformations, collecting metrics, etc). Usually the grammarware toolkit provides means to extend the grammar or rewrite parts of it — examples include TXL [12], GDK [24] and GRK [28]. Arguably the latter two of these examples also venture into the next category since they contain other grammar manipulation instruments like folding/unfolding. If we extend this arsenal with even more means like merging nonterminals, removing grammar fragments, injecting/projecting symbols from production rules, chaining/unchaining productions, adding/removing disjunctive clauses, permuting the order and narrowing/widening repetitions, we end up having an *operator suite* for grammar programming. The advantage of having such a suite lies in the simple fact that each of the operators can be studied and implemented in isolation, and the actual process of grammar programming will involve calling these operators with proper arguments in the desired order. Examples of operator suites include FST [29], XBGF [31], ΞBGF [46] and SLEIR [49].

2.3 Coupled transformation

We speak of coupled transformations when two or more kinds of mutually dependent software artefacts are transformed together to preserve consistency among them: usually one changes, and others co-evolve with it [27]. Naturally, the first coupled transformation scenario we should think of, involves an SPPF and a grammar that defines its structure. This change can be initiated from either side, let us consider both.

Assuming that we have a sequence of grammar transformation steps, we may want to execute them on the language instances (programs) as well, to make them compatible with the updated grammar. Such a need arises in the case of

grammar convergence [30], when a relationship between two grammars is reverse engineered by programming the steps necessary to turn one into the other, and a co-transformation can help to migrate instances obtained with one grammar to fit with the other. For example, we could have a grammar for the concrete syntax and a schema for serialisation of the same data — a transformation sequence that strips the concrete grammar from elements not found in the schema (typically terminals guiding the parsing process such as semicolons and brackets), could also be coupled with a transformation sequence that removes the corresponding parts from the graphs defined by them (e.g., a parse tree and an XML document).

Consider another scenario where we have the change on language instances and want to lift it to the level of language definitions. An example could be found in program transformation, a common software engineering practice of metaprogramming. If we want a refactoring like extracting a method, renaming a variable or removing a go-to statement, it is easy and practical to express it in terms of matching/rewriting paradigm: in Spoofax [20], Rascal [22], TXL [12], ATL [19], XSLT [21], etc. However, a correct refactoring should preserve the meaning of the program, and the first step towards that is syntactic correctness of this program. For non-refactoring transformations found in aspect-oriented development, automated bug fixing and other areas, we still want to ascertain the extent to which the language is extended, reduced or revised. In the case of strongly typed metaprogramming languages, they will not allow you to create any ill-formed output, but the development process can lead you to *first* specify a breaking transform and *then* cotransform the grammar so that it "fits" — which is what coupled transformations are good for.

2.4 Explicit versus implicit

This was already mentioned before, but becomes a crucial point from now on: parse forests can arise from two different sources — conjunctive clauses in the grammar used for parsing and generalised parsing with ambiguous grammars. The latter case can be considered implicit conjunction, since it is present on the level of language instances but not on the grammar level. In that case, instead of a more cumbersome construction specifying a precise parse, we use a simpler grammatical definition which yields a forest. If a grammar is both conjunctive and ambiguous, this can lead to its both implicit and explicit conjunctive clauses to be found in SPPFs — with no observable difference on an instance level.

Similarly, some of the transformations will "collapse" conjunctions, making one branch of a clause equal to another. Formally, for an SPPF node to have several branches means existence of several edges in the form $\langle v_i, (v_{i1}, ..., v_{ik_i})\rangle$, $\langle v_i, (v'_{i1}, ..., v'_{ik'_i})\rangle$, etc. When a transformation results in all v_ij becoming equal to the corresponding v'_{ij}, such edges merge in the set. If such conjunctions represent ambiguities, this is disambiguation; if they represent parse views, it merges the views and makes them undistinguishable.

	Language preserved	Language extended	Language reduced	Language revised
SPPFs preserved	bypass eliminate introduce import vertical horizontal designate unlabel anonymize deanonymize renameL renameS	addV addH define		
SPPFs preserved or fail			removeV removeH undefine	
SPPFs refactored	unfold fold inline extract abridge detour unchain chain massage distribute factor deyaccify yaccify equate rassoc lassoc renameN clone concatT splitT	appear widen upgrade unite removeC	disappear	abstractize project concretize permute renameT splitN
SPPFs refactored or fail			addC narrow downgrade	redefine replace reroot
fail				inject

Fig. 3. The XBGF operator suite designed for convergence experiments [30,31,51] and updated here to the latest version of the GrammarLab. Columns of the table refer to the effects of the operators on the string language generated by a grammar; rows classify coupled effects on the SPPFs.

3 Grammar-based gardening

XBGF (standing for "transformations of BNF-like grammar formalism") was an operator suite for grammar programming originally developed for grammar recovery and convergence experiments [30,31] and used for various grammar maintenance tasks afterwards — e.g., for improving the quality and maturity of grammars in the Grammar Zoo [47,50]. It has operators like **eliminate**(n) that checks whether the given nonterminal n is referenced anywhere in the grammar, and if not, removes its definition harmlessly; or operators like **removeN**(x, y) that ensures that the nonterminal x is found in the grammar while y is not, and subsequently renames x to y; or even operators like **redefine**(p_k, p'_k) which removes all production rules p_k defining one nonterminal from the grammar and replaces them with rules p'_k defining the same nonterminal differently. These operators are relatively well-studied so that we can always make a claim about the effect that a transformation chain has on the language generated/accepted by the grammar. Originally [45, §7] XBGF operators were classified according to their preservation, increase, decrease or revision of the language within two semantics: the string semantics and the term semantics. The contribution of this section is their classification according to the coupled effect of the operators on the SPPFs — see Figure 3 for the overview.

3.1 SPPFs preserved

The best kind of coupled transformation is the trivial one where the initial transformation triggers no change in the linked artefacts.

3.1.1 Language-preserving operators

Many operators that preserve the (string) language associated with the grammar, also preserve the shared packed parse forests of the instances of this language. Consider, for instance, the **eliminate**(n) operator we have just introduced in the previous paragraph: essentially, it removes an unused construct. Since such a construct is unused in other production rules, it can never be reached from the root symbol, so it can also never occur in the graphs representing grammatically correct programs. Hence, any SPPF which was correct for grammar before the transformation, is still correct for the grammar with the unused part eliminated. Similarly, introducing a language construct that was not previously there and is not (yet) linked to the root, has no impact on the forests. The same argumentation holds for decorating operators that add/remove labels to/from rules of the grammar or their subexpressions, or rename them.

The last two operators seen in this cell on Figure 3 are **vertical**(n) and **horizontal**(n) — they facilitate switching between a horizontal style of grammatical definitions (i.e., "`A ::= B | C;`") and a vertical one (i.e., "`A ::= B; A ::= C;`") — some grammatical frameworks distinguish between them, but never on an instance level, since a realisation of a disjunction commits to one particular branch. Hence, these operators also have no impact on SPPFs.

3.1.2 Language-extending operators

In the same way rearranging alternatives in production rules discussed in the previous section, has no impact on SPPFs, strict language extension operators like **addV**(p) and **addH**(p) have no impact on the forests. Since disjunctive clauses are not explicitly visible in SPPFs, any tree or forest derived with the original grammar, also conforms to the transformed one — the coupled instance transformation is trivial.

There is even one operator which is very invasive on a grammar level while being entirely harmless on the instance level — **define**(p) is a variant of **introduce**(p) that adds a definition of a nonterminal that *is used* in some parts in the grammar reachable from the top symbol. Having such nonterminals (called "bottom nonterminals") in a grammar is not a healthy practice and is in general considered a sign of bad quality since it signals incompleteness [26,41,47]. However, if we assume for the sake of simplicity that the default semantics for an undefined nonterminal is immediate failure (or parsing, generation, recognition or whatever the goal we need the grammar for), we may view **define**(p) as a language-extending (not a language-revising) operator. Thus, if we do somehow obtain a well-formed SPPF for such a grammar, it means it was constructed while *avoiding* the bottom nonterminal in question — hence, introducing it is no different than adding any other unreachable part we have seen so far and as such has no effect on the SPPFs.

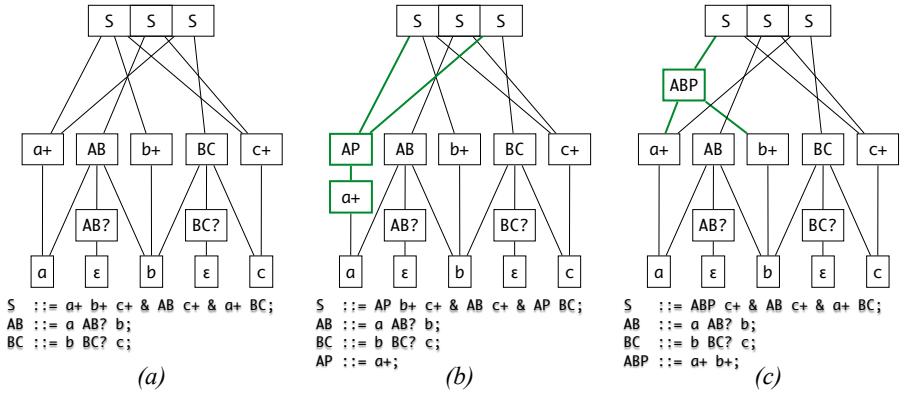


Fig. 4. SPPF transformations coupled with extraction of a new nonterminal definition: **(a)** the original grammar and an SPPF of the term "`abc`"; **(b)** after applying **extract**(`AP::=a+;`); **(c)** after applying **extract**(`ABP::=a+ b+;`). The case of extracting one symbol is easier because an SPPF already has nodes for such derivations and it only needs to be chained; when more than one symbol is present in the right hand side of a production rule being extracted, then a new node is introduced for all matched patterns of use.

3.2 SPPFs preserved, if possible

There are several cases when we do not know in advance whether the cotransformation of SPPFs will be possible: when it is, it is trivial.

3.2.1 Language-reducing operators

The operators **removeV**(p) and **removeH**(p) are the counterparts of **addV** and **addH** operators we have considered above, which remove alternatives instead of adding them. The effect of such a transformation on a given SPPF is easy to determine: if the alternative which is being removed, is exercised anywhere in the graph, the (co)transformation fails; if it is not, then no update of the forest is required.

Note that since all branches of the conjunctive clause are present in a given SPPF, their removal requires a (possibly failing) refactoring: hence, **removeC**(p) is considered later in §3.3.2.

The **undefine**(n) operator takes a valid nonterminal (defined and used within the grammar) and turns it into a bottom nonterminal (used yet not defined). It is a language reducing operator since its effect is a strict decrease in the number of possible correct programs: any parse graph containing a note related to the nonterminal n, becomes invalid. Hence, the coupled transformation for it checks whether such a node is indeed found in the given SPPF: if yes, the transformation fails; if not, it immediately succeeds without updating the SPPF.

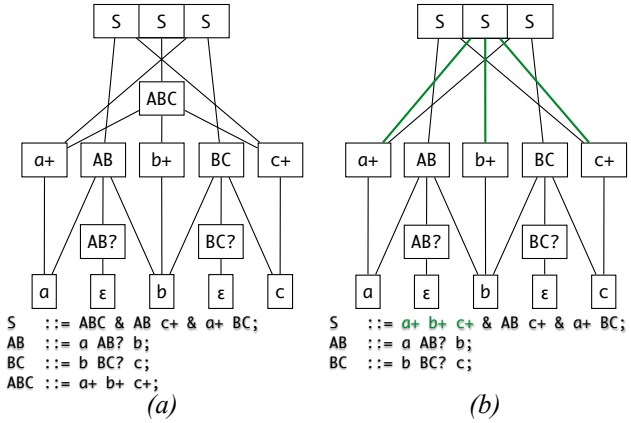


Fig. 5. SPPF transformations coupled with inlining a nonterminal definition: **(a)** the original grammar and an SPPF of the term "`abc`"; **(b)** after applying **inline**(`ABC`). The inlining is fairly straightforward: the node in question is removed, and any previously incoming edge is replaced with the list of previously outgoing edges.

3.3 SPPFs refactored

In the next subsections we consider cases of less trivial coupled transformations, when language instances have to change to preserve conformance.

3.3.1 Language-preserving operators

Many transformation operators that preserve the language associated with a grammar, still have some impact on the parse graphs. When the impact is easy to calculate in advance and thus encode the coupled transformations as SPPF refactorings that are parametrised in the same way the grammar transformations are, we can run commands like **extract**(p) on both grammars and SPPFs.

Consider Figure 4(a). It shows a simple grammar of a non-context-free language $\{a^n b^n c^n \mid n > 0\}$ with three conjunctive views: the first one (`a+ b+ c+`) being the most intuitive and hence the most suitable for expressing patterns to be matched on programs; the remaining two being used to parse the language (which is well-known to be context-sensitive, so we *need* the power of two conjuncts to recognise it precisely). In a sense, the last two conjuncts represent a recogniser and the first one specifies a parser [40,42]. When a transformation command **extract**(`AP::=a+;`) is executed, the effect on the grammar is apparent: a new nonterminal is introduced and two occurrences of its right hand side are replaced with it. The effect on an SPPF is also quite easy to calculate: the node with **a**+ is replaced with a chain of two nodes (`AP` and `a+`); the incoming edges of the old node are connected as the incoming edges to the first one in the chain; the outgoing edges of the old node become the outgoing ones of the last in

the chain (shown on Figure 4(b), changes in bold green). A slightly more complicated case is shown on Figure 4(c), where a new vertex needs to be created when we **extract**(`ABP::=a+ b+;`) because a symbol sequence `a+b+` did not correspond to any vertex in the old graph. For all vertices that had outgoing edges to both `a+` and `b+`, they got replaced by one edge to the new node. Figure 5 shows the opposite scenario of inlining a nonterminal in a grammar, coupled with "inlining" corresponding vertices in a graph by drawing edges through it.

Many other operators of this category from Figure 3 work similarly: **chain** replaces a node with a chain of two nodes; **fold** does the same folding we have seen above with **extract**, but without introducing a new nonterminal and possibly in a limited scope; **rassoc** and **lassoc** replace an iterative production rule with a recursive right/left associative one and thus stretches a node with multiple children into an unbalanced binary subtree; **concatT** and **splitT** merge or unmerge leaves, etc.

3.3.2 Language-extending operators

Above we have considered grammar transformation operators that add disjunctive clauses to the grammar, obviously extending the associated language. In the case of extended context-free grammars (regular right hand side grammars) that allow metasyntactic sugar like optionals ($x?$ effectively meaning $x|\varepsilon$) and regular closures (x^+ for transitive and x^* for reflexive transitive), the **widen**(e, e') operator is used to transform $x?$ to x^* or x to x^+, together with the **appear**(p) operator that transforms ε to $x?$ (effectively injecting an optional symbol). The coupled graph transformations for these cases usually boil down to inserting new vertices in the right places in order to keep the structural commitments up to date with the changed grammar.

An even less trivial case of language extension is called "upgrading" and involves replacing a nonterminal by an expression that can be reduced to it. For instance, in `A ::= B C; D ::= B|E;` we can upgrade `B` in `A` (underlined) to `D`. Such a transformation increases the string language associated with a grammar, as well as rearranges the relations between nonterminals. The coupled transformation for SPPF is still simple and inserts an extra vertex for `D` between `A` and `B` (`E` is still not present in the SPPF).

The **removeC**(p) operator that eliminates a conjunct, formally also increases the underlying language since any extra conjunct is possibly an extra condition to be met, and dropping it makes the combination weaker. Technically the coupled SPPF transform that removes a conjunct is a disambiguation filter, but it is not useful to count it as such since the ambiguity being removed is explicit (recall §2.4).

3.3.3 Language-reducing operators

The **disappear**(p) operator is used to transform $x?$ or x^* to ε. The coupled transformation on SPPFs for it exists, but is completely different from the ones being considered so far: it is inherently irreversible since if the SPPF in question

actually contains the x? with x as a child node, then that x is removed and lost. This is contrasting to folding/unfolding vertices and rearranging the edges around them.

3.3.4 Language-revising operators

The operators **abstractize**(p) and **concretize**(p) eliminate and introduce terminals from production rules (a common practice when mapping abstract syntax to concrete syntax, hence the names). Since the terminals are present explicitly in the arguments, we can easily implement our coupled SPPF transformations by inserting leaves and connecting them to the appropriate places to the graph, or removing them. These transformations can have a big effect on the SPPF and are therefore more similar to the coupled transform from the previous paragraph. The **project** operator is a stronger version of **abstractize** or **disappear** that works on any symbol, but the transformation coupled with it, is the same: locate all the parts being removed from the grammar, remove them from the graph.

The rest of language revising operators are coupled with less invasive rearrangements of the parse graph: reordering edges (**permute**), updating the contents of the leaves (**renameT**) and splitting one nonterminal into several (**splitN**).

3.4 SPPFs refactored, when possible

Cotransformations from the previous section were necessary but could never fail: they were applicable to all possible graphs. Let us now move on to cotransformations that could seem successful on the grammar level but fail on the instance level (causing the combination to fail).

3.4.1 Language-reducing operators

The **narrow** operator (the reverse of **widen** discussed above) and the **downgrade** operator (the reverse of **upgrade**) become simple parse graph rearrangements, if the constructs in the SPPF happen to correspond to the new grammar, and fail otherwise. For instance, if a "wider" option is found in the SPPF, we have no automated way to update it.

The **addC** operator, on the other hand, shows us yet another class of coupled transforms: namely, the one requiring reparsing. Indeed, if the first branch of the conjunctive clause of `S` from Figure 4 were to be introduced as a transformation step, we would need to reconnect the left subnode of `S` to the appropriate children, which formally corresponds to parsing. In the current prototype implementation we reuse the existing parser — to the best of our knowledge, other frameworks like TXL [12] do the same — instead of exploring possibly more efficient alternatives.

3.4.2 Language-revising operators

The most brutal among language revising operators: **redefine** that replaces an entire nonterminal definition with a different one; **replace** doing the same for arbitrary subexpressions; **reroot** that changes the starting symbol of the grammar, — all require reparsing as a part of their coupled transformation steps.

3.5 Cotransformations destined to fail

Interestingly, there is one particular operator that is always doomed: **inject**(p) that works like **appear** but can insert any symbol anywhere in the grammar. In order to construct a coupled SPPF transformation for **inject**, we need to know how to connect the new node to its children, but this information is ultimately lacking from the operator parameters. The only cases where it could have worked, are already covered by other operators (e.g., injecting terminals is **concretize**, injecting possibly empty symbols is **appear**).

4 Related and future work

As said before, we are not the only ones trying to use computation models based on graphs instead of trees in software language engineering. It remains to be seen whether systematically using abstract syntax graphs [36] and general purpose graph transformation frameworks would be much different. In that case grammars can also be represented as graphs similar to Wirth's syntactic charts [32].

Our approach to couple instance transformations to grammar transformations and not vice versa has its counterparts in other technological spaces such as modelware [17] or XML [25] or databases [18], obviously with transformations of metamodels or schemata as the starting point. Coupled transformations in general have been re-explained to some extent in this paper, but there is a much more detailed introduction [27].

Grammar mutations [46,49] are systematic generalisations of grammar transformations used for this paper. There does not seem to be any fundamental problem in combining that generalisation with our couplings, but the implementation of coupled mutations remains future work.

The classification of coupled SPPF transformations from Figure 3 corresponds to the two kinds of negotiated evolution: "adaptation through tolerance" when SPPFs are preserved and "through adjustment" when they are refactored [48].

There is a lot of work on disambiguation, parse forest pruning and shaving, and it remains to be seen whether our approach can usefully complement similarly-minded techniques from that area such as van den Brand et al [6]'s implementation of disambiguation filters with term rewriting.

SPPF transformations could possibly be represented formally as classical graph replacement systems that rewrite nodes [15] or (hyper)edges [13]. One of the main objectives of presenting this paper at the workshop is to estimate potential usefulness of this approach.

5 Conclusion

In this paper, we have considered coupled transformations of grammars together with shared packed parse forests defined by these grammars. An implementation of a transformation operator suite was proposed. Each grammar change was coupled to one of the following: (1) no change in the parse graphs; (2) rearranging the graphs; (3) introducing new elements to graphs based on operator arguments; (4) reparsing; (5) imminent failure. This classification is complementary to the previously existing ones based on preserving, increasing, reducing or revising the semantics chosen for the grammar.

The examples given in the paper mostly refer to concrete grammars in the context of parsing, but the research was done with software language engineering principles, which means that the contribution is applicable to coupled evolution of grammars as well as ontologies, API, DSLs, XML schemata, libraries, etc. We have used Boolean grammars as the underlying formalism due to their power to represent non-context-free languages, ambiguous generalised parses and parse views in a uniform way. This is the first project involving coupled transformations of Boolean grammars.

The computation model proposed in this paper, can be used for formalisations and proofs of certain properties of transformation chains; for grammar-based convergence; for manipulating parse views and in general for tasks involving synchronous consistent changes to Boolean grammars and shared packed parse forests. This is an area of rapidly growing interest in the software language engineering community, and its limits, as well as the extent of its usefulness, remains to be examined.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison-Wesley (1985)
2. Bagge, A.H., Lämmel, R.: Walk Your Tree Any Way You Want. In: ICMT. LNCS, vol. 7909, pp. 33–49. Springer (June 2013)
3. Bagge, A.H., Zaytsev, V.: Open and Original Problems in Software Language Engineering 2015 Workshop Report. SIGSOFT Software Engineering Notes 40(3), 32–37 (May 2015)
4. Basten, H.J., Vinju, J.J.: Parse Forest Diagnostics with Dr. Ambiguity. In: SLE'11. LNCS, vol. 6940, pp. 283–302 (2011)
5. van den Brand, M.G.J., de Jong, H.A., Klint, P., Olivier, P.A.: Efficient Annotated Terms. Software: Practice & Experience 30(3), 259–291 (2000)
6. van den Brand, M.G.J., Klusener, A.S., Moonen, L., Vinju, J.J.: Generalized Parsing and Term Rewriting: Semantics Driven Disambiguation. ENTCS 82(3), 1–17 (2003)
7. van den Brand, M.G.J., Scheerder, J., Vinju, J.J., Visser, E.: Disambiguation Filters for Scannerless Generalized LR Parsers. In: CC. pp. 143–158 (2002)
8. van den Brand, M.G.J., Heering, J., Klint, P., Olivier, P.A.: Compiling Language Definitions: The ASF+SDF Compiler. ACM ToPLaS 24(4), 334–368 (2002)

9. van den Brand, M.G.J., Klint, P., Vinju, J.J.: Term Rewriting with Traversal Functions. ACM ToSEM 12(2), 152–190 (Apr 2003)
10. Cantor, D.G.: On the Ambiguity Problem of Backus Systems. Journal of the ACM 9(4), 477–479 (1962)
11. Chomsky, N.: On Certain Formal Properties of Grammars. Information and Control 2(2), 137–167 (1959)
12. Dean, T.R., Cordy, J.R., Malton, A.J., Schneider, K.A.: Grammar Programming in TXL. In: SCAM. IEEE (2002)
13. Drewes, F., Kreowski, H.J., Habel, A.: Handbook of Graph Grammars and Computing by Graph Transformation. chap. Hyperedge Replacement Graph Grammars, pp. 95–162. World Scientific Publishing Co., Inc. (1997)
14. Earley, J.: An Efficient Context-Free Parsing Algorithm. Ph.D. thesis, Carnegie-Mellon University (Aug 1968)
15. Engelfriet, J., Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation. chap. Node Replacement Graph Grammars, pp. 1–94. World Scientific Publishing Co., Inc. (1997)
16. Ford, B.: Parsing Expression Grammars: a Recognition-Based Syntactic Foundation. In: POPL (Jan 2004)
17. Gîrba, T., Favre, J., Ducasse, S.: Using Meta-Model Transformation to Model Software Evolution. ENTCS 137(3), 57–64 (2005)
18. Henrard, J., Hick, J., Thiran, P., Hainaut, J.: Strategies for Data Reengineering. In: WCRE. pp. 211–220. IEEE (2002)
19. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., Valduriez, P.: ATL: a QVT-like Transformation Language. In: OOPSLA. pp. 719–720. ACM (2006)
20. Kats, L.C.L., Visser, E.: The Spoofax Language Workbench. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) SPLASH/OOPSLA. pp. 237–238. ACM (2010)
21. Kay, M.: XSL Transformations (XSLT) Version 2.0. W3C Recommendation (23 January 2007), http://www.w3.org/TR/2007/REC-xslt20-20070123
22. Klint, P., Storm, T.v.d., Vinju, J.: RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In: Proceedings of SCAM. pp. 168–177. IEEE Computer Society (2009)
23. Klint, P., Visser, E.: Using Filters for the Disambiguation of Context-Free Grammars. In: Pighizzini, G., Pietro, P.S. (eds.) Proceedings of the ASMICS Workshop on Parsing Theory. pp. 1–20. Universitá di Milano (1994)
24. Kort, J., Lämmel, R., Verhoef, C.: The Grammar Deployment Kit. In: LDTA. ENTCS, vol. 65. Elsevier (2002), 7 pages
25. Lämmel, R., Lohmann, W.: Format Evolution. In: RETIS. vol. 155, pp. 113–134. OCG (2001)
26. Lämmel, R., Verhoef, C.: Semi-automatic Grammar Recovery. Software—Practice & Experience 31(15), 1395–1438 (Dec 2001)
27. Lämmel, R.: Transformations Everywhere. Science of Computer Programming (SCP) 52, 1–8 (2004)
28. Lämmel, R.: The Amsterdam Toolkit for Language Archaeology. In: ATEM'04. ENTCS, Elsevier (2005)
29. Lämmel, R., Wachsmuth, G.: Transformation of SDF Syntax Definitions in the ASF+SDF Meta-Environment. In: LDTA. ENTCS, vol. 44. Elsevier (2001)
30. Lämmel, R., Zaytsev, V.: An Introduction to Grammar Convergence. In: Integrated Formal Methods (iFM). LNCS, vol. 5423, pp. 246–260. Springer-Verlag (Feb 2009)
31. Lämmel, R., Zaytsev, V.: Recovering Grammar Relationships for the Java Language Specification. Software Quality Journal (SQJ) 19(2), 333–378 (Mar 2011)

32. Martynenko, B.: Towards the 80th Anniversary of N. Wirth: Wirth's Syntactic Charts in the SYNTAX-Technology. In: SoRuCom. pp. 199–206. IEEE (Oct 2014)
33. Okhotin, A.: Conjunctive Grammars. Journal of Automata, Languages and Combinatorics 6(4), 519–535 (2001)
34. Okhotin, A.: Boolean Grammars. Information and Computation 194(1), 19–48 (2004)
35. Okhotin, A.: Conjunctive and Boolean Grammars: The True General Case of the Context-Free Grammars. Computer Science Review 9, 27–59 (2013)
36. Oliveira, B.C.d.S., Löh, A.: Abstract Syntax Graphs for Domain Specific Languages. In: PEPM. pp. 87–96. ACM (2013)
37. Parr, T., Fischer, K.: LL(*): the Foundation of the ANTLR Parser Generator. In: PLDI. pp. 425–436. ACM (2011)
38. Scott, E., Johnstone, A.: Generalized Bottom Up Parsers with Reduced Stack Activity. Computer Journal 48(5), 565–587 (2005)
39. Scott, E., Johnstone, A.: GLL Parsing. ENTCS 253(7), 177–189 (2010), LDTA'09
40. Scott, E., Johnstone, A.: Recognition is not Parsing — SPPF-style Parsing from Cubic Recognisers. Science of Computer Programming (SCP) 75(1-2), 55–70 (2010)
41. Sellink, M.P.A., Verhoef, C.: Development, Assessment, and Reengineering of Language Descriptions. In: CSMR. pp. 151–160. IEEE (Mar 2000)
42. Stevenson, A., Cordy, J.R.: Parse Views with Boolean Grammars. Science of Computer Programming (SCP) 97(1), 59–63 (2015), Special Issue on New Ideas and Emerging Results in Understanding Software
43. Tomita, M.: Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems. Kluwer Academic Publishers (1985)
44. Visser, E.: Scannerless Generalized-LR Parsing. Tech. Rep. P9707, University of Amsterdam (Jul 1997)
45. Zaytsev, V.: Recovery, Convergence and Documentation of Languages. Ph.D. thesis, Vrije Universiteit, Amsterdam, The Netherlands (Oct 2010)
46. Zaytsev, V.: Language Evolution, Metasyntactically. EC-EASST; BX 49 (2012)
47. Zaytsev, V.: Grammar Maturity Model. In: Pierantonio, A., Tamzalit, D., Schätz, B. (eds.) Ninth Workshop on Models and Evolution (ME 2014). pp. 42–51 (2014)
48. Zaytsev, V.: Negotiated Grammar Evolution. JOT; XM 13(3), 1:1–22 (Jul 2014)
49. Zaytsev, V.: Software Language Engineering by Intentional Rewriting. EC-EASST; SQM 65 (Mar 2014)
50. Zaytsev, V.: Grammar Zoo: A Corpus of Experimental Grammarware. Science of Computer Programming (SCP) 98, 28–51 (Feb 2015)
51. Zaytsev, V.: Guided Grammar Convergence. In: Poster proceedings of SLE'13 (2015), in print, available from CoRR: http://arxiv.org/abs/1503.08476
52. Zaytsev, V., Bagge, A.H.: Parsing in a Broad Sense. In: MoDELS. LNCS, vol. 8767, pp. 50–67. Springer (Oct 2014)

Conditions, constraints and contracts: On the use of annotations for policy modeling

Paolo Bottoni[1], Roberto Navigli[1], and Francesco Parisi Presicce[1]

Università di Roma "Sapienza" (Italy)
(bottoni,navigli,parisi)@di.uniroma1.it

Abstract. Organisational policies express constraints on generation and processing of resources. However, application domains rely on transformation processes, which are in principle orthogonal to policy specifications and domain rules and policies may evolve in a non-synchronised way. In previous papers, we have proposed annotations as a flexible way to model aspects of some policy, and showed how they could be used to impose *constraints* on domain configurations, how to derive application *conditions* on transformations, and how to annotate complex patterns. We extend the approach by: allowing domain model elements to be annotated with collections of elements, which can be collectively applied to individual resources or collections thereof; proposing an original construction to solve the problem of annotations remaining *orphan*, when annotated resources are consumed; introducing a notion of *contract*, by which a policy imposes additional pre-conditions and post-conditions on rules for deriving new resources. We discuss a concrete case study of linguistic resources, annotated with information on the licenses under which they can be used. The annotation framework allows forms of reasoning such as identifying conflicts among licenses, enforcing the presence of licenses, or ruling out some modifications of a licence configuration.

1 Introduction

Organisational policies express constraints on generation and processing of resources which are accepted by agents subject to, or anyway acknowledging, the authority of such organisations. However, agents retain the ability to operate on such resources according to their own strategies, as long as the results of these operations conform to the policy, or are used in areas not subject to it.

A typical example is that of licenses, which define ways in which software resources can be manipulated and made available for usage by third parties. The diffusion of open data [2] and of public repositories allows a dissemination of resources which can be employed in different ways, ranging from their simple replication for integration in local pools, to the creation of sophisticated services. While non proprietary resources can usually be accessed without restrictions, access to the results of resource manipulations could be subject to restrictions related to the safeguard of intellectual property. However, it is usually the case that the usage of certain resources in the construction of the service prevents

the possibility of imposing such restrictions, or forces specific forms of access compatible with those for the used resource.

In [5, 6] we proposed the usage of annotations as a flexible way to indicate the aspects for which domain model elements can fall under some policy, and showed how annotations could be used to impose *constraints* on configurations of domain resources, how to derive application *conditions* on their transformations, and how to annotate complex patterns. In particular, this allows the management of situations in which policies of access change, or different policies have to be simultaneously applied. Annotations are indeed a way of flexibly associating elements of different domains, and a number of techniques have been developed to express the constraints which are imposed on transformations modeling the evolution of elements in an *application* domain, when they are subject to constraints depending on annotations with elements of a *contextual* domain.

In this paper we extend the notion of annotation in four directions: first we propose an original construction for transformations of annotated models, which solves the problem of *orphan annotations*, which remain dangling when annotated resources are consumed. Second, we allow domain model elements to be annotated with *collections* of elements from the annotations domain, which can be collectively applied to individual elements or collections thereof. Third, we consider violations of constraints induced by the creation of elements for which an annotation must be provided and define *constraint repair actions* to solve such situations. Finally, we introduce a notion of *contract*, by which a policy imposes additional pre- and post-conditions on rules for deriving new resources. Again, the use of contracts leads to an original notion of transformation under contracts. In all the considered cases, the definition of transformations relies on the specific features of annotations and classical categorial constructions.

Paper organisation. After discussing related work in Section 2 and recalling fundamental notions in Section 3, we provide a construction for avoiding orphan annotations in Section 4 and introduce a motivational case study concerning linguistic resources annotated with licenses in Section 5. The model for rewriting under annotation constraints and contracts, illustrated via the case study, is presented in Section 6. Section 7 concludes the paper.

2 Related Work

A number of approaches have considered the management of inconsistencies in the field of graph transformations. In particular, mechanisms for ensuring that invariants are maintained throughout transformations have lead to the identification of mechanisms for the generation of pre- or post-application conditions to be associated with rules, or for manipulation of the left-hand or right-hand side of a rule. This topic was started in [12] and extensively explored in [13]. In [5, 6], we have shown how the separation of the application and contextual domains allows some simplified constructions for such a generation.

Another line of research involves the identification of inconsistencies of a model with respect to some desired (or undesired) property established at the

metamodel level. For example, conformance to a pattern can be imposed by completing the missing required parts of the pattern by a co-limit construction [4], required ordering for refactorings can be established by analysis of their conflicts [16], explicit transformations can be associated with guidelines to repair violations [1]. In this paper we consider only inconsistencies involving annotations, again leveraging domain separation.

Contracts were introduced in the form of pairs of graphs indicating pre- and post-conditions of operations [14]. In this way they define rule schemes which are typically instantiated by setting some parameters [8]. Based on this, one can consider satisfaction of conditions for service composition [17] or generate tests against these schemes to check correctness of model evolutions [22]. Our usage of contracts allows the definition of multiple contracts for a single rule and provides a basis for enforcing correctness on a local basis, as opposed to corrections required by violations of global constraints.

The need for a formalisation of licenses has been addressed for services, where service composition has to take into account that different services may run under different licenses, as part of service level agreement, focusing mostly on service behaviour [10]. As for data, work on the definition of ontologies for the management of digital rights have been conducted by Garcia *et al.* [11] and Rodríguez-Doncel et al. [20]. Graph transformations have been employed in the closely related field of access control, where techniques similar to the ones employed here were used (see e.g. [15]).

3 Preliminaries

Graphs and morphisms. We introduce classical notions from graph transformation theory (see [7]).

Definition 1. *A* graph *is a tuple* $G = (V_G, E_G, s, t)$*, where* V_G *is a set of* nodes*,* E_G *is a set of* edges*,* $s\colon E_G \to V_G$ *and* $t\colon E_G \to V_G$ *are the* source *and* target *functions, i.e. graphs are considered to be* directed*.*

Definition 2. *Given two graphs* G_1 *and* G_2 *a* morphism $\mu\colon G_1 \to G_2$ *is a pair of functions* $\mu_V\colon V_{G_1} \to V_{G_2}$*,* $\mu_E\colon E_{G_1} \to E_{G_2}$ *such that* μ_E *preserves the images of sources and targets, i.e. for* $e \in E$ *we have:* $s_2(\mu_E(e)) = \mu_V(s_1(e))$ *and* $t_2(\mu_E(e)) = \mu_V(t_1(e))$*.*

We use morphisms for a number of purposes, some of which exemplified through Figure 1.

Typing. For G_1 a graph, and G^T a *type graph*, $\mu\colon G_1 \to G^T$ is a *typing morphism* if μ is total. Typed graphs are extended to attributed typed graphs defining specific sets of attributes for them. A node of a certain type will be associated with a value for each of the attributes defined by the type.

Transformation rule. One (as in the Single Pushout Approach, SPO, see Figure 1 (c)) or two (as in the Double Pushout Approach, DPO, see Figure 1 (a))

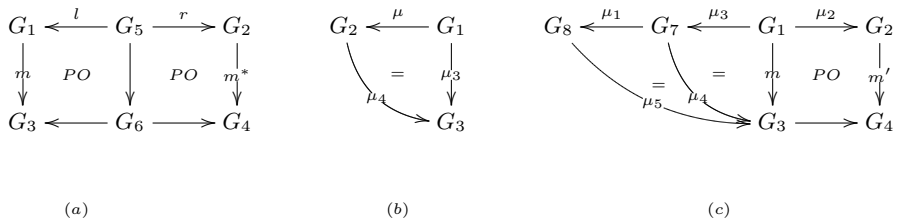


Fig. 1. (a) DPO Derivation, (b) satisfaction of constraint, (c) SPO derivation with AC.

type-preserving morphisms are used to define rules. Regardless of the form, a rule p identifies a graph G_1 and the applicability of p to a graph G_3 depends on the existence of a total type-preserving morphism $m\colon G_1 \to G_3$. If a rule p is applied to a graph G_3 to produce graph G_4, we write $(G_3, G_4) \in \Longrightarrow_p$.

Constraint. $\mu\colon G_1 \to G_2$ defines a *constraint* together with a satisfaction relation, $\models$: for any graph G_3, $G_3 \models \mu$ *iff* for each morphism $\mu_3\colon G_1 \to G_3$, there exist a morphism $\mu_4\colon G_2 \to G_3$ such that the triangle of Figure 1(b) commutes[1]. A *negative* constraint, denoted by $\neg\mu\colon G_1 \to G_2$, is satisfied by G_3 if no such morphism μ_4 exists for some μ_3. The particular case $\neg\mu\colon G_1 \to G_1$, defines a *forbidden graph* and is represented by the single graph G_1.

Application condition. Given a (DPO or SPO) rule p with identified graph G_1, $\mu_1\colon G_7 \to G_8$ is an *application condition* (AC) for p if it restricts the relation $\Longrightarrow_p$ to pairs $(G3, G_4)$ for which there exist morphisms $\mu_3\colon G_1 \to G_7$, $\mu_4\colon G_7 \to G_3$ and $\mu_5\colon G_8 \to G_3$ such that the triangles in the diagram of Figure 1(c) commute (in the SPO approach the square in it is a pushout, analogously for the two squares in Figure 1(a)). The requirement that no such μ_4 exist is called a negative application condition (NAC).

All of the above is naturally extended to sets of rules and to attributed type graphs, so that constraints and application conditions can refer to prescribed values that attributes of the matched nodes must have. Rules can require updates on the values associated with preserved nodes, or assignment of values for the created nodes. In all these cases,we follow the approach of symbolic attributed graphs [19], using constraint satisfaction to evaluate conditions and attributes.

Annotations. Annotations of nodes of a domain $\mathcal{D}_1$ with nodes of a domain $\mathcal{D}_2$ are defined via nodes of types derived from `AnnotationNodeType`. We call $\mathcal{A}$ the domain of such annotation nodes. Each annotation node $a \in \mathcal{A}$ participates in exactly one instance of the pattern $\pi_a = x \xleftarrow{e_1} a \xrightarrow{e_2} y$, where $x \in \mathcal{D}_1$, $y \in \mathcal{D}_2$, e_1 is an edge of an application-dependent type and e_2 is an edge of type `annotatesWith`. We work on type graphs TG resulting from the disjoint union of the type graphs defining $\mathcal{D}_1$ (TG_1) and $\mathcal{D}_2$ (TG_2), together with the relevant annotation node and edge types, and we consider two types of constraints related to annotations, assuming that all the constraints on the application domain are

[1] G_1 is also called P, from *premise*, and G_2 is also called C, from *conclusion*.

preserved by the domain rules. Constraints of the first type are derived from π_a and are of the form $\mu : \boxed{A} \rightarrow \boxed{X \xleftarrow{e_1} A \xrightarrow{e_2} Y}$, for X some node type in TG_1, Y some node type in TG_2, A an annotation node type and e_1 and e_2 as described before, with possible restrictions on the association of X and Y. Well-formedness results from conformance with the disjunction of all such constraints. Given a type graph TG we call $\mathcal{L}_{\pi_a}(TG)$ the language of all the graphs in TG which are well-formed with respect to the individual domains and the annotation pattern. The second type of constraints expresses specific policies via morphisms of the form $\mu : G_1 \rightharpoonup G_2$ where G_1 is typed on the TG_1 component of TG, G_2 is typed on TG with well-formed annotations, and its projection on TG_1 is isomorphic to G_1. We call these *annotation constraints*. In Section 6 we will also introduce *contracts* based on morphisms involving two graphs typed on TG. All the rules in the paper are typed on TG_1, and all the constraints are annotation constraints.

4 Annotations and collections

Definition 3 from [6] extend graphs and morphisms to include the notion of *box*.

Definition 3. *A (directed)* graph with boxes *(or* B-graph*) is a tuple* $G = (V, E, B, s, t, cnt)$*, where: (1)* V *and* E *are as in Definition 1; (2)* B *is a set of* boxes*, such that* $B \cap (V \cup E) = \emptyset$*; (3)* s *and* t *extend their codomains to* $V \cup B$*; (4)* $cnt : B \rightarrow \wp(V \cup B)$ *is a function associating a box with its* content[2] *with the property that if* $x \in cnt(b_1)$ *and* $b_1 \in cnt(b_2)$*, then* $x \in cnt(b_2)$*.*

A *type B-graph* includes a set of box types B^T which are sources or targets for edge types. Moreover, a function $cnt^T : B^T \rightarrow \wp(V^T \cup B^T)$ associates each type of box with the set of types of elements it can contain. Similarly, a (total) morphism on B-graphs was defined in [6] by adding a component μ_B, preserving the content function, i.e. for all $x \in V \cup B, b \in B$, one has $x \in cnt_1(b) \implies \mu_{V \cup B}(x) \in cnt_2(\mu_B(b))$, with $\mu_{V \cup B}$ the (disjoint) union of μ_V and μ_B. In [6] total morphisms allowed DPO transformations of B-graphs.

Based on these notions, annotation nodes can be used not just with reference to individual nodes in TG_1 and values in TG_2, but also to collections in either domain, i.e. a collection of annotation values can be used in an atomic way to annotate some resource or collection thereof. However, in the original formulation of rewriting with boxes in [6], removing an annotated element from a model $G' \in \mathcal{L}_{\pi_a}(TG)$ would create dangling annotation edges, forbidding rule application in the DPO form. Nor can deletion in the SPO form be employed. Even if in this case the annotation edges would be removed, we would be left with *orphan annotations*, i.e. annotation nodes to which no annotation edge is attached, resulting in a graph not in $\mathcal{L}_{\pi_a}(TG)$. Hence, we devise an original mechanism, based on Construction 1, to complement a transformation performed according to the DPO appproach in TG_1 so that well-formedness is preserved.

Construction 1. With reference to Figure 2, let its top two rows depict the usual DPO application of a rule $L \leftarrow K \rightarrow R$ to a graph G obtained by applying to G'

[2] Here and elsewhere $\wp$ denotes the powerset.

the morphism $\mathbf{f_1}$ induced by the forgetful functor given by the restriction of TG to TG_1. The unique morphism induced by the immersion of the image of $\mathbf{f_1}(G')$ into G' is also derived. Then D' is the unique (up to isomorphisms) graph in $\mathcal{L}_{\pi_a}(TG)$, i.e. with well-formed annotations, which is maximal with respect to the occurrences of the annotation pattern and for which the left square at the bottom of the diagram in Figure 2 is a pullback (hence its restriction to the elements typed on TG_1 is isomorphic to D). Finally, H' is given by the pushout of H and D' along D.

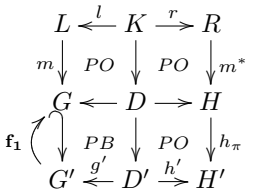


Fig. 2. Avoiding orphan annotations.

Figure 3 illustrates Construction 1 with a model example where teams in an organisation temporarily gather members for specific tasks [6]. G' describes a configuration where `frank` - a member, together with `paul`, of the `softEng` team - is also the only member of the `security` team. For both `frank` and `security`, time annotations indicate that they can operate within the organisation only at daytime. The DPO rule at the top, removing a team with only one member, is applied to G, the projection according to f_1 of G', by identifying `security` and `frank` with `1:Team` and `2:Member` in L, respectively. As a consequence, the annotation on `security` and the edges touching it are removed, updating the *cnt* function accordingly, while the one for `paul` is preserved.

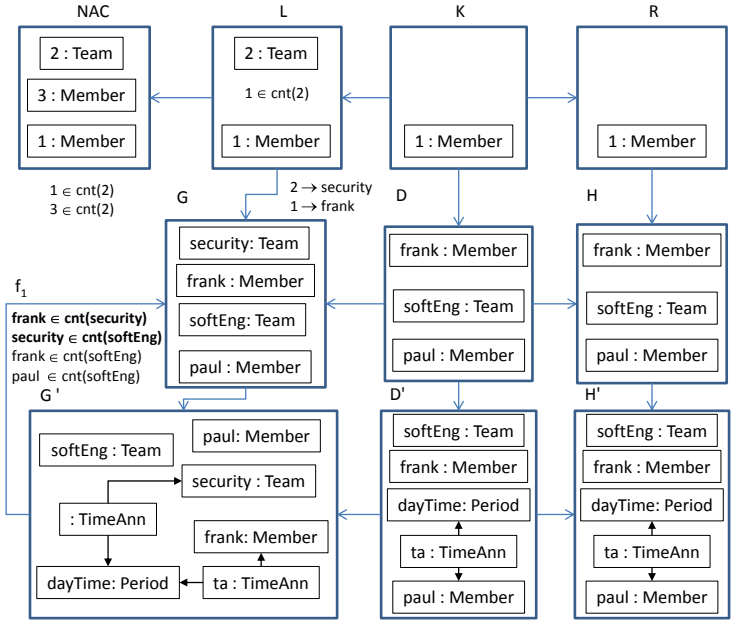


Fig. 3. An example of Construction 1 for the removal of an annotated team.

Note that Construction 1 applies to removal of the application domain elements, not of contextual ones. Actually, we assume here that contextual domains are fixed. The correctness of Construction 1 is expressed by Proposition 1.

Proposition 1. *For $G' \in \mathcal{L}_{\pi_a}(TG)$ and a rule $L \leftarrow K \rightarrow R$, the graph H' generated as in Construction 1 is in $\mathcal{L}_{\pi_a}(TG)$. Moreover, $Ann(H') \subseteq Ann(G')$, where $Ann(X)$ denotes the set of annotation nodes in a graph X.*

Proof. We first observe that $Ann(H') = Ann(D')$, so that $Ann(H') \subseteq Ann(G')$, since H, which is typed on TG_1, does not present any new annotation. Since D' is well-formed, none of its annotation nodes is an orphan.

5 Case study: licenses

A *license* specifies admissible forms of access, usage and redistribution of resources and of the results of manipulating them. While individual resources can be associated with specific licenses, the licenses connected to a resource usually depend on some policy associated with the repository from which it is extracted. Moreover, such repositories often publish resources under a number of licenses, which must be all simultaneously respected, and which are normally transferred to the extracted resources. As this imposes some form of compatibility among licenses, deciding whether a certain usage is admitted may become complex. As an example, elements published under a `Creative Commons`, `CC`, scheme can be associated with combinations of the following licenses: `NC` for `NonCommercial` (the resource cannot be used for commercial purposes), `BY` for `Attribution` (credit to the author is acknowledged), `SA` for `ShareAlike` (derived resources must be redistributed preserving the original licenses), and `ND` for `NoDerivatives` (remixing, transforming, or building upon the resource may not grant redistribution). Each element is considered to be of `PublicDomain` (`PD`).

BabelNet[3] [18] is a multilingual semantic network containing millions of concepts (e.g. the **apple fruit** concept) and named entities (e.g. the **Apple Inc.** entity), interlinked via semantic relations. A single node in the network is called a Babel *synset* and contains a set of synonyms which express a given concept or named entity in different languages. For instance, the **apple fruit** concept is represented by the synset {*apple_EN*, *pomme_FR*, *mela_IT*, ..., *manzana_ES*}. A word occurring in a synset is called *sense* (e.g., *apple_EN* in the above synset is the fruit sense of the ambiguous word apple).

BabelNet itself is obtained as the result of the automatic mapping (which is in turn considered a kind of resource unique to BabelNet) and integration of several, publicly available, knowledge repositories. However, each repository provides resources (e.g. synsets and senses) under different licenses. For example, WordNet [9], Wikidata[4] and parts of the OMWN project [3] are released under a permissive license, which allows any use, either commercial or non-commercial, of the data;

[3] http://babelnet.org
[4] http://wikidata.org

Wikipedia and Wiktionary are released with a `CC-BY-SA` license; OmegaWiki and other wordnets in OMWN are released under `CC-BY`; the Basque wordnet in OMWN is released under `CC-BY-NC-SA` and so is the BabelNet mapping between all these resources. Unfortunately, not all of the licenses are compatible with one another, as shown in the compability chart for CC licenses in Table 1, where ✓ indicates compatibility, × incompatibility, and ! that usage is not recommended. For instance, Wikipedia, whose license is `CC-BY-SA`, cannot be merged with data from the Basque wordnet or the BabelNet mapping. Interestingly, some mergings can be done in one direction only, e.g. from a resource in a `CC-BY` repository, such as OmegaWiki, one can derive a new resource with a more restrictive `CC-BY-SA` license, thereby making it compatible with, e.g., Wikipedia.

Table 1. The compatibility chart for Creative Commons licenses.

Compatibility chart		Terms that may be used for a derivative work or adaptation						
		BY	BY-NC	BY-NC-ND	BY-NC-SA	BY-ND	BY-SA	PD
Status of original work	PD	√	√	√	√	√	√	√
	BY	√	√	√	√	√	√	!
	BY-NC	!	√	√	√	!	!	!
	BY-NC-ND	×	×	×	×	×	×	×
	BY-NC-SA	×	×	×	√	×	×	×
	BY-ND	×	×	×	×	×	×	×
	BY-SA	×	×	×	×	×	√	×

However, the opposite is not possible, as no-one can modify a `SA` license. As a result, some licenses, such as `CC-BY-NC-SA` and `CC-BY-SA`, are inherently and mutually incompatible. To solve this problem, BabelNet is viewed as a collection of knowledge resources with heterogeneous licenses. For instance, it is possible to consider a subset of resources which can be used commercially, e.g. Wikipedia, WordNet and others. However, the resources with `NC` license (e.g. the Basque wordnet and the BabelNet mapping) cannot be used commercially. As the mapping is the enabling technology for interconnecting the various resources into a whole unified, multilingual network, any commercial use is subject to obtaining a commercial license from the BabelNet's authors.

5.1 A model of linguistic services and licenses

To represent the management of licenses and languages in BabelNet, we model synsets, senses, etc. as nodes (or boxes) of specific types from a domain, $\mathcal{R}$, of *Resources*. Similarly, licenses are modeled as nodes of type `License`, from the domain of *Licenses*, $\mathcal{L}$, with an attribute `name` ranging over strings identifying the different kinds of license, and languages are modeled as nodes of type `Language`, from the domain of *Languages*, $\mathcal{K}$, with `name` ranging over the available languages. We consider $\mathcal{R}$ as the application domain, and $\mathcal{L}$ and $\mathcal{K}$ as

contextual domains, typing the annotation edges accordingly. Thus, for `xxx` a type in $\mathcal{R}$, edges of types `xxxLicAnnotation` and `xxxLangAnnotation` allow its annotation with elements of $\mathcal{L}$ and $\mathcal{K}$, through edges of type `annotatesWith`.

Figure 4 presents the type graph TG for BabelNet. Stereotypes indicate whether an element is of the *Node* or *Box* sort, and if it comes from the domain $\mathcal{R}$, $\mathcal{L}$ or $\mathcal{K}$, or is an `AnnotationTypeNode`. In the domain $\mathcal{R}$, a `Sense` is characterised by an attribute `content`, with values in the sort of strings. Moreover, the three types of box, `Synset`, `Collection` and `LicenseBundle`, can contain only elements of suitable types, namely `Sense`, `Synset` and `License`, respectively. The attribute `concept` for `Synset` indicates the concept represented by that collection of senses. The type `LicenseBundleAnn`, of the *Node* sort and derived from `AnnotationTypeNode`, can be used to relate resources with `LicenseBundle`. However, as it inherits from `LicenseAnn`, one can annotate any element in the resource domain with single licenses or with license bundles. A `Request` to *obtain* a `Collection` can be annotated with both license and language information (not indicated to avoid cluttering) and *activates* a `Service` *producing* the collection.

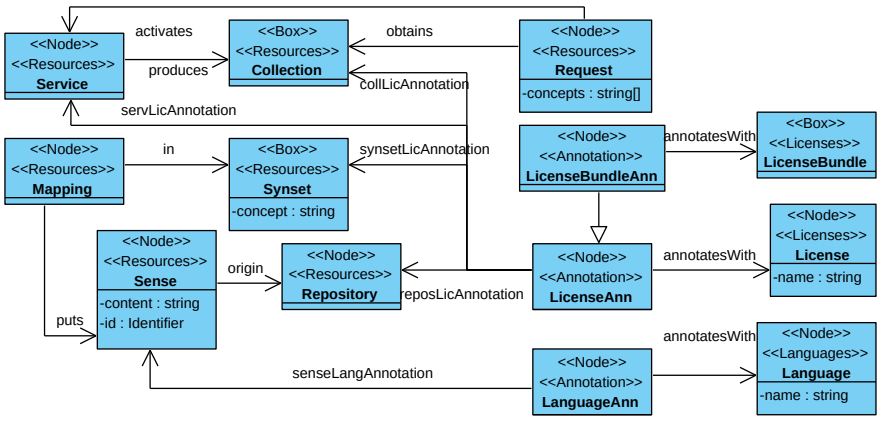


Fig. 4. The type graph for modeling the running example

We first model the basic working of BabelNet services via increasing rules in the *Resources* domain. Rule `createCollection` in Figure 5 (top) creates an initially empty collection for the synsets to be served in response to a query to define the concepts in a set Z. Two rules allow the inclusion in the collection of a synset for a concept in the request. Rule `addSynset` in Figure 5 (middle) is used to add an already available synset in the collection if it is not already there, as indicated by the NAC. The other one, not shown, creates and adds a synset for a concept, if it does not exist already. Rule `createMapping` in Figure 5 (bottom) is used to populate synsets with senses representing the concept, according to a mapping, with a NAC to avoid including a sense twice.

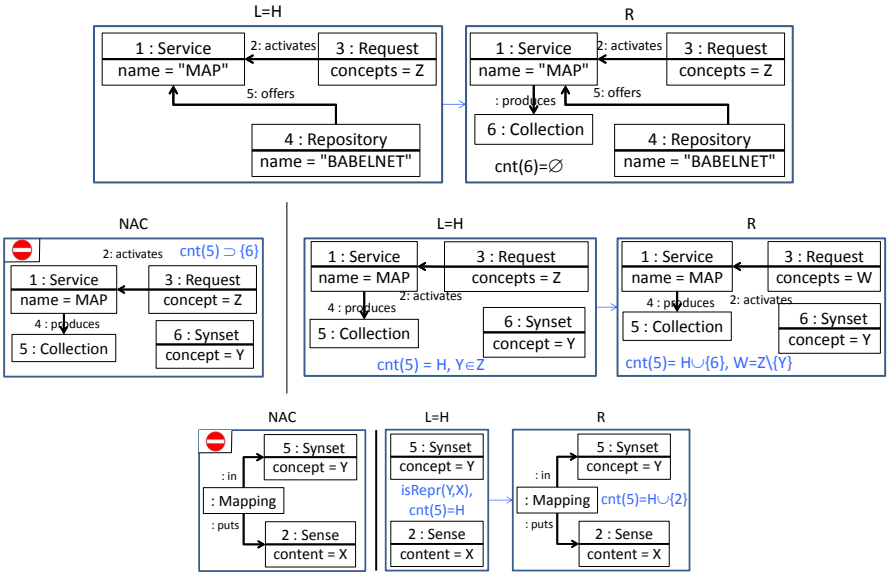


Fig. 5. Rule `createCollection` prepares a container for serving a request (top). Rule `createSynset` sets up a synset for a requested concept (middle). Rule `createMapping` relates a sense to a synset (bottom). Vertical lines separate NACs from rule morphisms.

In all these cases, the bundle of licenses associated with the invoked service must be associated with the produced collection, as will be discussed in Section 6.2. Moreover, requests can be further characterised, for example by annotating them with particular licenses or languages, so that only senses annotated with those licenses or languages are included in the obtained collection, as per suitable application conditions, following the constructions in [5].

6 Maintaining consistency with constraints and contracts

While the result of applying a rule is guaranteed to produce a correctly typed graph, it might be the case that such a graph does not conform to further conditions imposed on the model at hand. We identify two dimensions along which conditions can be distinguished: one pertaining to the identification of the domain for which the condition is defined (whether the application domain or the domain resulting from the annotation process) and one relative to the scope of the condition (whether global to the domain or local to specific transformations).

Concerning the latter dimension, we use constraints, as introduced in Section 3, to impose well-formedness conditions for a domain. We assume that the host graph to which a rule is applied is well-formed, and we identify mechanisms to ensure that the transformation process produces another well-formed graph.

In [6], we have presented some constructions to derive application conditions for rules from annotation constraints. By leaving the rule morphism alone, we maintain a form of separation of concerns, making rule reuse simpler when different forms of annotation are involved. Intuitively, those constructions work when a rule adds elements related to elements matched by the left-hand side, but the annotation constraint imposes these relations to exist only between elements annotated in specific ways. An application condition ensures that such an annotation context already exists in the host graph when the rule is applied.

In this section we focus on situations in which the addition of application conditions is not sufficient, since the required context cannot already exist, in particular if a newly created element must be annotated in specific ways. This would require the right-hand side of a rule to be enriched with the appropriate annotation, but then the rule would no longer be defined only on the application domain. To approach this problem, we consider separately situations violating global constraints, and situations violating conditions on specific rules, that we model as contracts. Since we are interested in rules which add new elements, i.e. $L = K$ for a DPO rule, we present them as simple morphisms.

6.1 Management of constraints

As shown in Table 1, licenses can require or forbid the presence of one another. While the first case can be modelled by a positive constraint[5], we model the second via forbidden graphs. Figure 6 (left) shows the forbidden graph expressing that no resource can be annotated with both licenses `SA` and `ND`. An analogous graph will forbid the presence of both licenses in the same bundle. The constraint on the right requires that each resource be `PD`, where we use the generic type name `Resource` to refer to any of the types from the *Resource* domain.

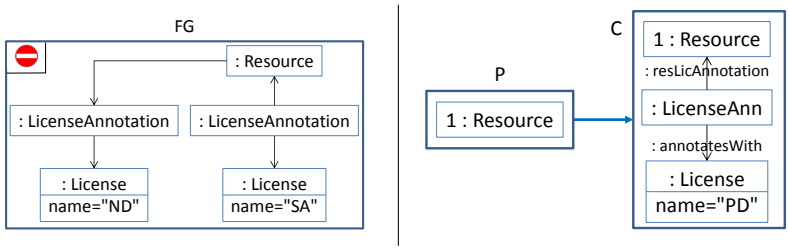


Fig. 6. A graph forbidding the simultaneous presence of licenses `SA` and `ND` (left) and a positive constraint assessing that each resource is `PD`.

The application of a rule may disrupt a constraint, typically by not creating the proper annotations. Hence, given a constraint μ, *constraint repair actions* are automatically inferred and applied, which modify the derivation relation so that the result satisfies μ.

[5] This would be a constraint with annotations both in P and C, not considered here.

Definition 4 (Constraint repair action). *Let $\mu_1 \colon G_1 \to G_2$ be a rule and $\mu_2 \colon G_3 \to G_4$ an annotation constraint. We define the relation $\Longrightarrow_{\mu_1,\mu_2}$ with reference to Figure 7. For any two graphs G_5 and G_6 such that $(G_5, G_6) \in \Longrightarrow_{\mu_1}$ as witnessed by the leftmost square, and $G_6 \not\models \mu_2$ (i.e. there exists a morphism $\mu_5^i \colon G_3 \to G_6$ but no morphism $\mu_6 \colon G_4 \to G_6$ for which the triangle formed by μ_2, μ_5^i and μ_6 commutes), we have $(G_5, G_7) \in \Longrightarrow_{\mu_1,\mu_2}$, where G_7 is constructed as the colimit of all the diagrams constructed by taking the pushout of μ_5^i and μ_2 along G_3 for each μ_5^i via the morphisms $\mu_7^i \colon G_4 \to G_7^i$ and $\mu_4^i \colon G_4 \to G_7^i$. The set of all the μ_4^i is called a* constraint repair action.

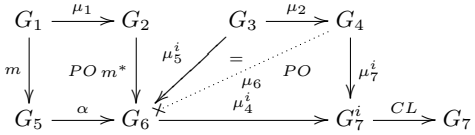


Fig. 7. Direct Derivation Diagram for a rule with constraint repair action.

Following Proposition 2 a repair action produces a graph compliant with μ_2.

Proposition 2. *Given G_7 and μ_2 as in Definition 4 we have $G_7 \models \mu_2$.*

Proof. We know that G_3 has a match in G_5, so that the constraint is violated by the presence of some element without proper annotation, which is added in each G_7^i as an effect of the pushout. Since we only deal with annotation constraints, each G_7^i does not present violations of μ_2 which were not in G_4, but actually presents one less violation. By taking the colimit, no violation appears in G_7.

Proposition 3 allows the cumulative application of repair actions.

Proposition 3. *Let G_6 be as in Definition 4 and $M_2 = \{\mu_2^j \colon G_3^j \to G_4^j \mid G_6 \not\models \mu_2^j\}$. Then, let G_7' the colimit of all the graphs G_7 constructed as in Definition 4 for each $\mu_2^j \in M_2$. Then $G_7' \models \mu_2^j$ for each $\mu_2^j \in M_2$.*

Proof. Since the premise of each μ_2^j does not contain annotation elements, no G_7 constructed for one constraint can add new violations of any other constraint. The result then follows by the associativity of colimits.

6.2 Contracts

Contracts define situations in which the application of a rule requires the production of some annotation, but only in situations in which elements in its left-hand side are associated with specific annotations.

Definition 5 (Contract). *Given a rule $\mu_2 \colon G_3 \to G_4$, a* contract on μ_2, *γ, is given by a morphism $\mu_1 \colon G_1 \to G_2$ (G_1 and G_2 typed on TG) together with spans $G_1 \overset{\mu_3}{\leftarrow} G_5 \overset{\mu_4}{\rightarrow} G_3$, $G_4 \overset{\mu_5}{\leftarrow} G_6 \overset{\mu_6}{\rightarrow} G_2$ (formed by total injective morphisms) and a morphism $\mu_7 \colon G_5 \to G_6$, (G_5 and G_6 typed on TG_1), such that all the closed paths in the upper part of Figure 8 commute.*

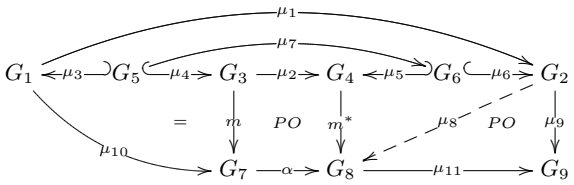


Fig. 8. Direct Derivation Diagram for a rule with contract enforcement action.

As an example, the top contract in Figure 9 describes the overall policy for license assignment: each collection is generated with the same collection of licenses of the generating service. Here, numbers are used to identify the nodes common to the rule and the contract, and letters for identifying elements related by the contract morphism.

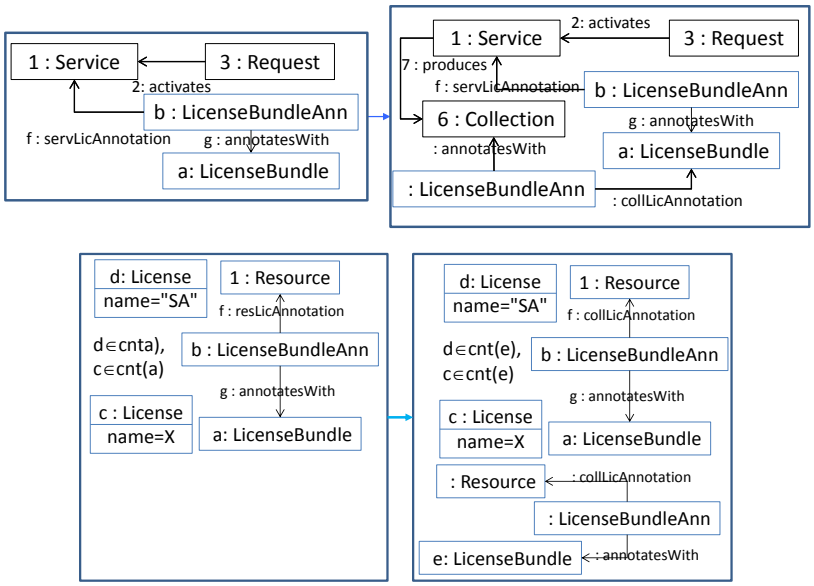


Fig. 9. A contract stating that each collection comes with the license of the service which generated it (top) and a contract specifically modeling the `SA` licence (bottom).

We can now model the asymmetry in license extension described in Section 5 with reference to the contract in Figure 9 (bottom): if a copy of a resource annotated with a bundle including the `SA` license is generated, the bundle annotating the new resource must preserve all the original licenses. This forbids the possibility of contracts which remove some license when `SA` is present, while it allows adding more licenses to the bundle, if not in contrast with others already present. Analogous contracts can be devised for multiple annotations with single

licenses, rather than with a bundle of licenses. It is important to note that the directionality inherent to this contract would not be expressible via constraints, which would either impose or forbid the presence of annotations in the bundles both before and after rule application.

The application of a domain rule creating new elements typically violates contracts requiring that they be annotated in certain ways. Hence, actions must be taken, as specified in Definition 6

Definition 6 (Contract enforcement action). *Let $\mu_2 \colon G_3 \to G_4$ and γ be as in Definition 5. A derivation $(G_7, G_8) \in \Longrightarrow_{\mu_1}$* fulfills *the contract γ* iff *for each morphism $\mu_{10} \colon G_1 \to G_7$ such that the leftmost square in the diagram of Figure 8 commutes, and for each morphism $\mu_{5_i} \colon G_6 \to G_4$ there exists at least one morphism $\mu_{8_i} \colon G_2 \to G_8$ so that the triangle $m^* \circ \mu_{5_i} \circ \mu_7 \colon G_5 \to G_8$, $\mu_1 \circ \mu_3 \colon G_5 \to G_2$ and $\mu_{8_i} \colon G_2 \to G_8$ commutes.*

If the above does not hold, the pair (G_7, G_8) is said to breach *the contract μ_1. A breach can be repaired by a* contract enforcement action *μ_{11} for μ_1, μ_2 on G_7 by constructing $G_8 \overset{\mu_{11}}{\to} G_9 \overset{\mu_9}{\leftarrow} G_2$ as the pushout of the span $G_8 \overset{m^* \circ \mu_{5_i}}{\leftarrow} G_6 \overset{\mu_{11}}{\to} G_2$. We denote the derivation thus obtained by $(G_7, G_9) \in \Longrightarrow_{\mu_2, \mu_1}$. Note that if no μ_{10} exists, then (G_7, G_8) also fulfills the contract.*

Whenever multiple contracts apply to μ_2, the final graph to be obtained for its application is represented by the colimit of all the diagrams thus formed, noting that in all such diagrams the pushout formed by $G_7 \leftarrow G_3 \to G_4$ and $G_7 \to G_8 \leftarrow G_4$ remains the same. Notice also that by a straightforward application of the associativity and commutativity of colimits, the same result can be obtained by successively applying the above construction to individual contracts, regardless of the order. The proof of Proposition 4 is then straightforward.

Proposition 4. *For a rule μ_2 and a contract μ_1 on it, each derivation in $\Longrightarrow_{\mu_2, \mu_1}$ fulfills μ_1.*

Theorem 1 states that applying the enforcement action is equivalent to applying a rule resulting from the composition of μ_2 and μ_1.

Theorem 1. *Given a rule $\mu_2 \colon G_3 \to G_4$ and a contract $\mu_1 \colon G_1 \to G_2$ on μ_2, there exists a rule μ_2' such that for each pair $(G_7, G_9) \in \Longrightarrow_{\mu_2, \mu_1}$, we have $(G_7, G_9) \in \Longrightarrow_{\mu_2'}$.*

Proof (Sketch.). Referring back to the diagram in Figure 8, the new left-hand side is constructed as the pushout of the span $G_1 \overset{\mu_3}{\leftarrow} G_5 \overset{\mu_4}{\to} G_3$, while the right-hand side as the colimit of the collection of spans $G_4 \overset{\mu_{5i}}{\leftarrow} G_6 \overset{\mu_6}{\to} G_2$. Shown in Figure 10 are the induced matching morphism $G_3' \to G_7$, the new rule $\mu_2' \colon G_3' \to G_4'$, and the result G_9 of applying μ_2' to G_7' via the induced matching. Since G_4' already "contains" G_2, there is no need for a subsequent enforcement action.

We can also define *negative contracts*, indicated as $\mu_1 : G_1 \overset{\neg}{\to} G_2$ such that $(G_5, G_6) \in \Longrightarrow_{\mu_1, \mu_2}$ only if $G_6 \not\models \mu_2$. This prevents the application, to the same μ_1, of other contracts of the form $\mu_1' : G_1' \to G_2'$ with $G_1' \hookrightarrow G_1$ and $G_2 \hookrightarrow G_2'$.

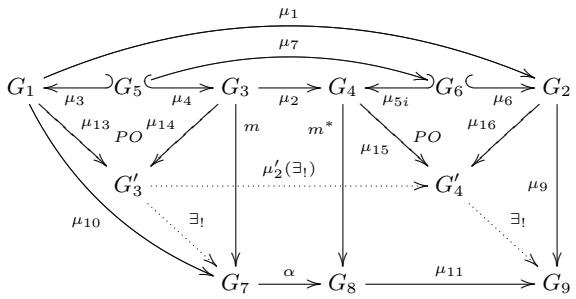


Fig. 10. Direct Derivation Diagram for composition of rule and contract.

The constructions above can be adapted to general, i.e. not only increasing, DPO rules $G_3 \leftarrow G_{10} \rightarrow G_4$ by considering contracts in the form of spans $G_1 \leftarrow G_{11} \rightarrow G_2$ and identifying the spans $G_5 \leftarrow G_{12} \rightarrow G_6$ and $G_{10} \leftarrow G_{13} \rightarrow G_{11}$.

In all these cases, we consider morphisms which are injective in the restrictions to $\mathcal{D}_1$ of the involved graphs, as well as in the immersion of $\mathcal{D}_1$ into $\mathcal{D}$, while they can be non-injective in the restriction to $\mathcal{D}_2$.

7 Conclusions and Future Work

We have extended the theory of annotation from [5, 6] by considering the problem of orphan annotations, left when annotated elements are deleted, and introducing contracts, relating pre- and post-conditions on the usage of annotated elements. We have used annotations to model the role of licenses in the open data environment defining the approved usages for resources. In this context, license bundles have been seen as a technique to manage sets of resources homogeneous with respect to the applicable licenses.

With respect to the theory, future research will involve considering constraints with annotations also in the premise, and to study dependencies and conflicts [21] within sets of contracts and within compositions of rules and contracts. Also, we want to generalise the notion of contract to that of contract schemes, allowing the customised generation of contracts for different rules.

As concerns the application domain considered in the paper, we plan to extend this work towards a generic framework for managing licenses, taking into consideration also other licensing schemes, by which declarative specifications of resource usages could be checked for verification of conformance to licenses. Moreover, the mentioned analysis tools in the field of graph transformations can be applied to verify the internal consistency of license bundles.

References

1. C. Amelunxen, E. Legros, A. Schürr, and I. Stürmer. Checking and enforcement of modeling guidelines with graph transformations. In *Proc. AGTIVE 2007*, volume 5088 of *LNCS*, pages 313–328. Springer, 2008.

2. S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. DBpedia: A nucleus for a web of open data. In *The Semantic Web*, volume 4825 of *LNCS*, pages 722–735. Springer, 2007.
3. F. Bond and K. Paik. A survey of wordnets and their licenses. In *Proc. GWC 2012*, pages 64–71, 2012.
4. P. Bottoni, E. Guerra, and J. de Lara. A language-independent and formal approach to pattern-based modelling with support for composition and analysis. *Information & Software Technology*, 52(8):821–844, 2010.
5. P. Bottoni and F. Parisi Presicce. Annotation processes for flexible management of contextual information. *JVLC*, 24(6):421–440, 2013.
6. P. Bottoni and F. Parisi-Presicce. Annotations on complex patterns. *ECEASST*, 58, 2013.
7. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.
8. G. Engels, M. Lohmann, S. Sauer, and R. Heckel. Model-driven monitoring: An application of graph transformation for design by contract. In *Proc. ICGT 2006*, volume 4178 of *LNCS*, pages 336–350. Springer, 2006.
9. C. Fellbaum, editor. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.
10. G. Gangadharan and V. DAndrea. Service licensing: conceptualization, formalization, and expression. *Serv. Orient. Comp. and Appl.*, 5(1):37–59, 2011.
11. R. Garca, R. Gil, and J. Delgado. A web ontologies framework for digital rights management. *Artificial Intelligence and Law*, 15(2):137–154, 2007.
12. A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundam. Inform.*, 26(3/4):287–313, 1996.
13. A. Habel and K.-H. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *MSCS*, 19(2):245–296, 2009.
14. J. H. Hausmann, R. Heckel, and M. Lohmann. Model-based development of web services descriptions enabling a precise matching concept. *Int. J. Web Service Res.*, 2(2):67–84, 2005.
15. M. Koch and F. Parisi Presicce. UML specification of access control policies and their formal verification. *Software and System Modeling*, 5(4):429–447, 2006.
16. T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Software and System Modeling*, 6(3):269–285, 2007.
17. M. Naeem, R. Heckel, F. Orejas, and F. Hermann. Incremental service composition based on partial matching of visual contracts. In *Proc. FASE 2010*, volume 6013 of *LNCS*, pages 123–138. Springer, 2010.
18. R. Navigli and S. P. Ponzetto. BabelNet: The automatic construction, evaluation and application of a wide-coverage multilingual semantic network. *Artificial Intelligence*, 193:217–250, 2012.
19. F. Orejas and L. Lambers. Symbolic attributed graphs for attributed graph transformation. *ECEASST*, 30, 2010.
20. V. Rodríguez-Doncel, S. Villata, and A. Gómez-Pérez. A dataset of RDF licenses. In *Proc. (JURIX) 2014*, pages 187–189. IOS Press, 2014.
21. O. Runge, C. Ermel, and G. Taentzer. AGG 2.0 - new features for specifying and analyzing algebraic graph transformations. In *Proc. AGTIVE 2011*, volume 7233 of *LNCS*, pages 81–88. Springer, 2012.
22. O. Runge, T. A. Khan, and R. Heckel. Test case generation using visual contracts. *ECEASST*, 58, 2013.

Global Graph Transformations

Luidnel Maignan and Antoine Spicher

University Paris-Est Créteil, LACL
61 avenue du Général de Gaulle
F-94015 Créteil Cedex, France
{luidnel.maignan,antoine.spicher}@u-pec.fr

Abstract. In this paper, we consider *Global Graph Transformations* where all occurrences of a set of predefined local rules are applied altogether synchronously so that each part of the original graph gives rise to a part of the result graph, without any reference to the original one. The particularity here is that our framework is deterministic. This is achieved by incorporating a notion of mutual agreement between its local rules. Our proposition is first motivated and illustrated on existing problems coming from different domains. It is then formalized as a categorical construction which is finally compared to more usual algebraic constructions, in particular to the strongly related Amalgamation Theorem. Applications of this work include the generalization of cellular automata and the clarification of some frameworks of complex systems modeling where the usual mutual exclusion of rule applications can be replaced by a concept of mutual agreement.

1 Introduction

The framework proposed herein has been designed with the need to model *deterministic* dynamical systems by graph transformations. The state of such a system is represented by a graph and its *global* dynamics is specified through a set of *local* evolution rules.

One such example is the simple framework of cellular automata (CA). The state of a CA is usually represented by a labeled regular graph where the nodes are the cells, the labels encode the cell states, and the edges represent the neighborhood relation between cells. A global evolution step consists of a synchronous update of all the labels relying on a local evolution function, the structure of the graph remaining unchanged. From another point of view, each patch of neighbor cells (e.g. triple of consecutive cells in 1D CA with radius 1, square of 9 cells in 2D Moore CA) gives rise to a new node with the updated label. All these new nodes are then connected together to form the next state graph representation which is independent from the current one. Recently, different formalizations based on this point of view have been proposed to generalize CA to arbitrary graphs with dynamic structures [1,2]. This paper is an attempt to show that an algebraic approach can provide an interesting alternative to those formalisms.

Other work was already devoted to the simulation of the so called *Dynamical Systems with Dynamical Structure* [5,20]. In these approaches, a rewriting of cell

complexes (an extension of graphs to higher dimensions, see Section 3) has been developed to simulate concurrent interaction rules. The resulting programming language, called MGS, has been shown as a unification of many computation models including CA, Lindenmayer systems, membrane computing systems, etc. So far, MGS parallel rule application strategies rely on a maximal-parallel property: only mutually exclusive matchings can be applied simultaneously. This leads to some non-determinism since there might be different maximal sets of mutually exclusive matchings. This paper arises as an attempt to model systems where the natural mutual agreement between the local rules applications can be used to overcome the difficulties introduced by the concept of mutual exclusion.

As a result of these two motivations, the transition mechanism of the modeled system is described in our setting by a set of rules that do not send rooted graphs to nodes as in CA, nor nodes to graphs as it is often the case in graph transformation [7], but that send graphs to graphs. Moreover, the reconstruction of the resulting global state from the local applications of these rules on the current state, relies on the following coherence property of the rules: when two rule matchings overlap on an input graph, the local behavior of the common part has to be shared by the two rules. This intuition can be seen as a compound of two instances of a simpler situation: any time a first matching includes a second matching, the result of the first matching has to include the result of the second matching. Because the proposed formalism is a direct expression of the coherence property, we believe that this framework allows to model very intuitively any desired deterministic system, and can be easily adapted to any particular need (*e.g.*, non-determinism, presence of terminals/non-terminals). Of course, as for any modeling framework, it certainly asks for some getting-used-to to users already accustomed with other modes of thinking.

The proposed framework reminds some existing ones. The statement of the coherence above is the same leading to the concept of sub-rule and amalgamation in the double-pushout approach [3]. It is also reminiscent of the connecting or gluing mechanisms used in node or edge based parallel graph grammars (see Sect. 2). This very simple inclusion intuition can be applied in various settings, as cell complexes in the following. In section 5, it is expressed categorically; this allows a short and intrinsic formalization with possible instantiations to different kinds of objects. Finally the evolution described by a total function in the CA setting simply becomes functors on a full subcategory in our setting.

Organization of the Paper. The rest of the paper is organized as follows. Section 2 provides some comparison with existing approaches of parallel graph transformation. Section 3 gives some formal preliminaries. Section 4 considers the example of triangular mesh refinement in order to expose the idea of the proposed framework informally. This example has been chosen because it encompasses many considerations about the proposed framework. Section 5 formalizes the concept of global transformation. It is then compared with the double-pushout approach and the strongly related concept of amalgamation of productions. Section 6 discusses the remaining aspect of the work and concludes.

2 Related Work

In this section, we describe briefly existing approaches of parallel rewriting systems with a final comparison to global transformations.

One of the first parallel rewriting systems are *Lindenmayer Systems* (LS), which are parallel string rewriting in contrast with sequential Chomsky grammars. A $\langle k, l\rangle$ LS ($k, l \in \mathbb{N}$) is a context-sensitive system gathering productions in a transition table, so that each sub-word of length $k + 1 + l$ (*i.e.*, a letter with a left context of length k and a right context of length l[1]) is associated with (possibly many) words. The parallel rewriting of a word is done as follows: each letter is substituted accordingly to its left and right contexts by one of its associated words in the transition table. Obviously, the LS is deterministic if the transition table is a function (*i.e.*, associates a unique word with each entry). LS have been extensively studied for their expressive powers and compared to the classic Chomsky hierarchy of formal languages [19]. They have also been used in the modeling of unidimensional and tree-like dynamical systems [16]. LS can be seen as a special case of parallel graph transformation, restricted to linear/sequential graphs. Other special cases of graphs can be addressed. For example, Paŭn Systems (roughly speaking, nested parallel multiset rewriting systems [15]) can be considered as complete graph transformations. Such systems derive naturally from LS by considering rewriting modulo associativity and commutativity: any two symbols of a string representing a multiset can be made neighbors by permuting the letters of the word. In this setting, a production is a metaphor of a chemical reaction. The left hand side (l.h.s.) of a production designates a sub-multiset which is entirely replaced by its associated right hand side (r.h.s.) (in contrast with LS where each letter is replaced independently). To avoid the consumption of the same symbol by two different reactions, the maximal-parallel strategy is considered leading to non-determinism.

Extension of LS to arbitrary graph transformation is not an easy task [7]. A first idea consists in encoding the graph using sets, multisets, sequences or terms, and their associated well-known and rich techniques. As an example, [17] bridges graph rewriting to set rewriting by considering a graph as a set of (hyper-)edges, an hyper-edge being a sequence of vertices. On the other hand, some works address the issue of a direct rewriting of graphs. Classical work in graph grammars includes node-rewriting and hyperedge-rewriting graph grammars [18]. These works have some interesting relation with our framework when they are used in a parallel setting, that is, when each node (resp. edge) chooses a production rule to apply. In this case, all of them provide a resulting graph and these graphs are connected (resp. glued) together in some way or another by an *embedding mechanism*. In the node setting, edges are used to specify the connection between the graphs while, in the hyperedge setting, the nodes specify the gluing between the graphs. In fact, any deterministic instances of these systems can easily be represented in our framework.

[1] An extra dummy symbol, the *marker*, is used to deal with boundaries.

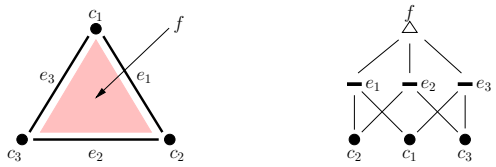


Fig. 1. On the left, a cell complex composed of three 0-cells (c_1, c_2, c_3), of three 1-cells (e_1, e_2, e_3) and of a single 2-cells (f). On the right, the Hasse diagram of its incident relationship.

The previous examples are set-theoretic approaches: graph transformations are expressed in terms of sets and set operations. Graph transformations can also be represented algebraically using the double-pushout and the simple-pushout approaches which formalize the idea of local replacement in a categorical manner [18]. The double-pushout approach is inherently local so that it needs to be extended to deal with parallel applications leading to the concepts of *parallelism* [4] and *amalgamation* [3]. Roughly speaking, parallelism allows to apply many mutually exclusive matchings simultaneously. Amalgamation provides a more general setting where the set of productions is augmented with sub-productions that handle some kind of overlaps. Therefore, many matchings can be applied together as long as sub-productions are chosen to deal with the overlapping sub-parts which is reminiscent of the coherence property stated above. Multi-amalgamation [6] is an extension of amalgamation to consider maximal matchings. However, the amalgamation theorem makes clear that a compound production can be applied only when each of its parts can be applied. This constraint makes (multi-)amalgamation unusable straightforwardly in the cases where the transformation of matchings only makes sense when taken altogether.

3 Formal Preliminaries and Notations

Since the present article follows naturally work introduced in [20], we consider *cell complexes*, a more general setting than graphs. Like a graph, a cell complex is a formal construction that builds a space in a combinatorial way through more basic objects called *topological cells*. Each topological cell abstractly represents a part of the whole and is characterized by a natural integer called *dimension*. A topological cell of dimension d is called d-cell. The structure of the whole space, corresponding to the partition into topological cells, is considered through the *incidence relationships*, relating two "neighbor" cells in the partition. A graph can then be seen as a cell complex composed of 0-cells and 1-cells, respectively the nodes and the edges, so that the incidence relationship of the complex coincides with the usual notion of incidence in graphs. More generally, a cell complex using only two dimensions is an undirected multi- and hyper-graph. There exist many possible formal definitions of cell complexes coming from different fields

(algebraic topology [12], digital topology [8], geometric modeling, etc.). We consider here the definition of [20].

Let $\mathcal{L}$ be an arbitrary set of symbols. A *labeled abstract cell complex* $\mathcal{K}$ *with labels in* $\mathcal{L}$ is given by a tuple $\langle \mathrm{C}_\mathcal{K}, \prec_\mathcal{K}, \dim_\mathcal{K}, \mathrm{l}_\mathcal{K} \rangle$ where $\mathrm{C}_\mathcal{K}$ is the set of *abstract topological cells*, $\prec_\mathcal{K}$ is a locally finite[2] strict partial order relation over $\mathrm{C}_\mathcal{K}$, $\dim_\mathcal{K} : \langle \mathrm{C}_\mathcal{K}, \prec_\mathcal{K} \rangle \to \langle \mathbb{N}, < \rangle$ is a strictly monotonic map assigning the dimension to each cell, and $\mathrm{l}_\mathcal{K} : \mathrm{C}_\mathcal{K} \to \mathcal{L}$ is a map assigning a label to each cell. An example of an abstract cell complex is shown on Fig. 1. We denote Acc the set of abstract cell complexes. In the following, we use the term *cell complex* (resp. *cell*) for abstract cell complex (resp. topological cell) since there is no possible ambiguity.

A *cell complex morphism* $h : \mathcal{K} \to \mathcal{K}'$ from a cell complex $\mathcal{K}$ to a cell complex $\mathcal{K}'$ is given by a strictly monotonic map $\mathrm{C}_h : \langle \mathrm{C}_\mathcal{K}, \prec_\mathcal{K} \rangle \to \langle \mathrm{C}_{\mathcal{K}'}, \prec_{\mathcal{K}'} \rangle$ such that $\mathrm{l}_\mathcal{K} = \mathrm{l}_{\mathcal{K}'} \circ \mathrm{C}_h$, and $\dim_\mathcal{K} = \dim_{\mathcal{K}'} \circ \mathrm{C}_h$. A *cell complex inclusion* $i : \mathcal{K} \to \mathcal{K}'$ from a cell complex $\mathcal{K}$ to a cell complex $\mathcal{K}'$ is a cell complex morphism from $\mathcal{K}$ to $\mathcal{K}'$ such that C_h is injective.

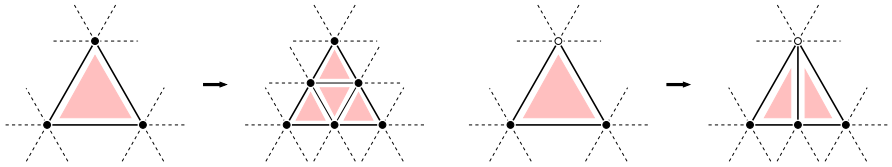
We consider the category $\mathbf{AccM}_\mathcal{L}$ of cell complexes and cell complex morphisms between them on one hand, and the sub-category $\mathbf{AccI}_\mathcal{L}$ of cell complexes and cell complex inclusions between them on the other hand, together with the associated inclusion functor $\mathrm{U}_\mathcal{L} : \mathbf{AccI}_\mathcal{L} \to \mathbf{AccM}_\mathcal{L}$. In the following, the subscript $\mathcal{L}$ is omitted, the set of labels is clear from the context and we never consider different sets of labels simultaneously. For the categorical discussion, we use the concepts of categories, functors, natural transformations, pushouts, colimits and comma categories. For formal definitions of these concepts, we refer to [10].

4 Specification of a Triangular Mesh Refinement

Mesh refinement is an approach used in geometrical modeling to generate smooth surfaces from an initial set of control points. Mesh refinement algorithms consist generally in iteratively generating new control points from current ones by applying a set of creation rules. Such procedures are commonly used in numerical resolution schemes and can be specified through graph transformations [14]. In [20], the declarative expression and implementation of these algorithms as a maximal-parallel cell complex rewriting are discussed.

Although mesh algorithms are intuitively described by local graphical schemes, they operate on the whole mesh and then turn out to be *global* and *synchronous*. Let us illustrate this issue by considering the Loop subdivision procedure [11] which is one of the simplest algorithms for refining triangular meshes (cell complexes where all 2-cells respect the incidence given in Fig. 1). It relies on the polyhedral subdivision where each triangle of the original mesh is substituted by four triangles as shown on Fig. 2a. This rule is quite informal. In particular, notice the dashed edges: they are definitively not part of the local transformation

[2] For any elements $x, y \in \mathrm{C}_\mathcal{K}$, the interval $[x, y] = \{ z \mid x \prec_\mathcal{K} z \prec_\mathcal{K} y \}$ is finite.



(a) Inner Rule (b) Boundary Rule

Fig. 2. Polyhedral Subdivision Scheme

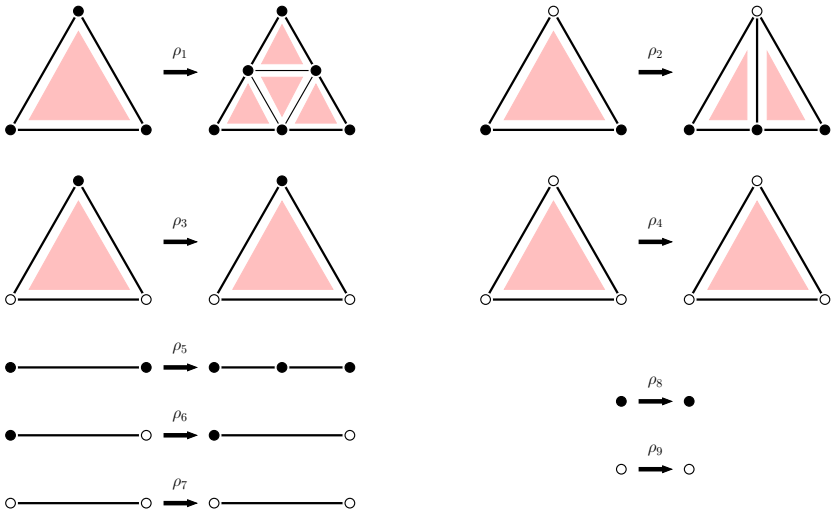
of the triangle but express how the new pattern has to be merged with the hypothetical applications of the same rule on some neighbor triangles. However, this detail —in the sense that the primary role of the rule is to specify the refinement of one triangle— has serious consequences on the algorithm since it forbids to consider the application of the rule on only one triangle and worse it only allows the transformation of the whole mesh. The reason of such a constraint comes from the considered class of cell complexes, *i.e.*, the mesh must remain triangular. This particularity makes mesh refinement an excellent candidate to illustrate our approach. In this section, we first specify a set of transformation rules for polyhedral subdivision, and then we detail how these rules are applied in the context of a global transformation.

4.1 Rules Specification

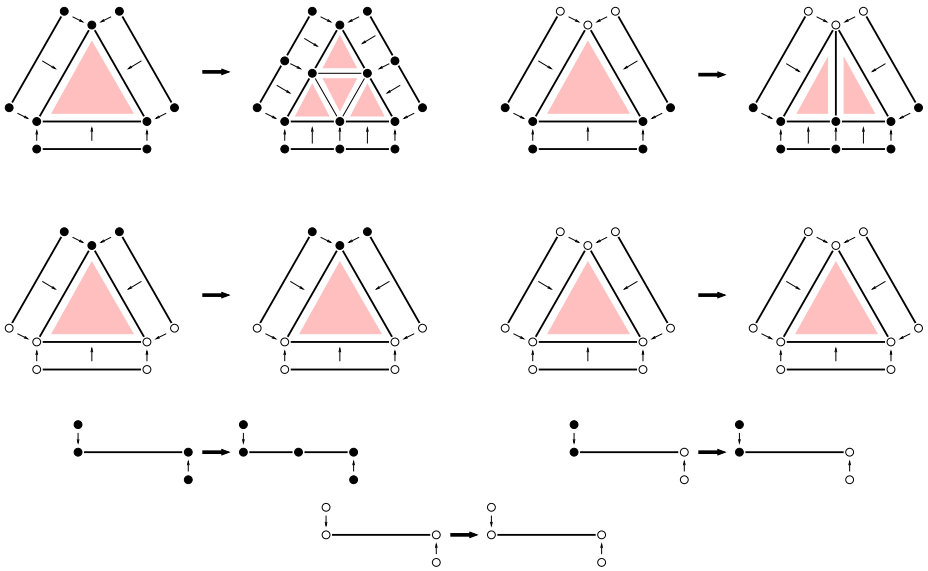
For the sake of illustration, let us consider that the refinement is restricted to a sub-part of the mesh. The region is specified by white or black labeled nodes so that the subdivision only occurs on triangles incident to three black nodes. Solutions exist to stop the natural propagation of the procedure over the whole mesh. Here, we consider an additional rule (see Fig. 2b) which deals with triangles located on the boundary of the region to be refined, *i.e.*, incident to exactly two black nodes. Other triangles are left unchanged.

The use of dashed contexts in Fig 2 is not accurate. Let us design a complete set of rules allowing the refinement of a triangular mesh in one global step. As a starting point, we specify the transformation of any single triangle of the mesh by considering two subdivision rules, ρ_1 and ρ_2 (corresponding to Fig. 2 without dashed context), and two more additional rules, ρ_3 and ρ_4, for unmodified triangles. These rules are shown on top of Fig. 3a.

Obviously, there is a lack of specification since connections between triangles are not taken into account. For instance, let us consider two triangles with only black nodes connected by a common edge as shown on the left of Fig. 4. Two matchings of the l.h.s. of rule ρ_1 are clearly identified. Therefore, the resulting cell complex should be composed of two instances of the r.h.s. of rule ρ_1. However, nothing specifies the way to built it. Forgetting mesh refinement for a moment, many possible constructions are conceivable: leaving the r.h.s. instances isolated,



(a) Local Transformation Rules



(b) Inclusion between Rules

Fig. 3. Polyhedral Subdivision Rules

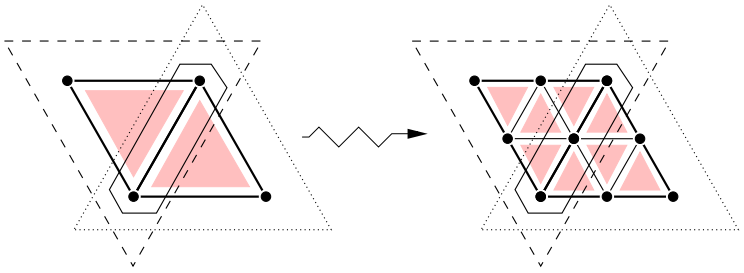


Fig. 4. Overlapping between two triangles

or merging them by a vertex or an edge, or even identifying them in only one instance, and so on and so forth.

In the case of mesh refinement, when two triangles share an edge, the two corresponding edges on the r.h.s. should also be shared on the resulting cell complex. There are two ways to specify such a behavior: one union-based method where a super-rule is designed for pairs of triangles, and another dual intersection-based method where an agreement is given about the transformation of the common edge. Here, we choose the latter method focusing on the evolution of an edge incident to two black nodes. Two things are required: firstly rule ρ_5 of Fig. 3a specifies the subdivision of that edge; secondly for each edge found on the l.h.s. of rule ρ_1, the associated divided edge is identified in the r.h.s. of ρ_1 as shown on Fig. 3b.

With this information at hand, the previous example of two side-by-side triangles is clarified: there are two matchings of ρ_1 and one matching of ρ_5 included in the two former matchings. The result cell complex is the unique cell complex where we can see two instances of the r.h.s. of ρ_1 and one instance of the r.h.s. of ρ_5 with inclusions between them being exactly the ones dictated by the inclusion rule. This can be observed in Fig. 4 where the inclusions between the l.h.s. matchings and the correspondence with the associated r.h.s. inclusions between the corresponding r.h.s. instances are depicted.

Iterating this design process for all possible intersections between rules, we obtain all the rules of Fig. 3a and all the inclusions given in Fig. 3b. Rules ρ_6 and ρ_7 describe the conservation of edges incident to a white node and the conservation of nodes is specified in ρ_8 and ρ_9. The latter are necessary for triangles sharing only one node. After transformation, their r.h.s. must also be connected by a node[3].

[3] As a byproduct of this intersection-based methodology, rules ρ_5-ρ_9 are used to manage isolated edges and nodes, which seems coherent with the refinement process. If these effects are not desired then the union-based methodology is applicable, leading to more transformation rules, namely 32 pair-rules since there are 4 types of triangles and two triangles are either incident by an edge or a node. This allows to stay strictly concerned with triangles.

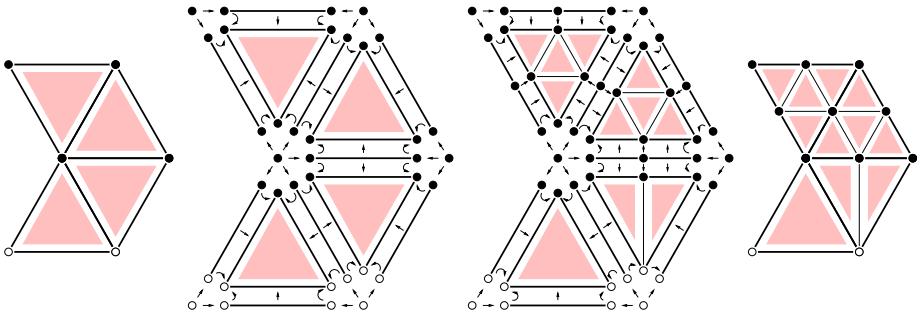


Fig. 5. Application Steps

As for the rule ρ_5, all inclusions between the l.h.s.'s of these new rules are found and the associated r.h.s.'s are identified. This association of inclusion has to be seen as a set of rules too, that we call *inclusion rules*: for each inclusion of one l.h.s. in another l.h.s., an inclusion rule specifies an associated inclusion between the two corresponding r.h.s. Put in another way, theses inclusion rules express a kind of *mutual agreement* between the rules. When a rule indicates a transformation on a l.h.s., and this l.h.s. includes the l.h.s. of another rule, the former rule must achieve the transformation required by the latter rule, and this is materialized by an inclusion rule indicating where this required transformation can be found. So no mutual exclusion is necessary. This is reminiscent of the notion of sub-production in amalgamation. The comparison with this concept is delayed to Sect. 5.3.

4.2 Rules Application

The construction procedure described above about the transformation of two side-by-side triangles can be generalized to any mesh as follows:

1. *Pattern matching* (Fig. 5a ⇒ 5b): the original mesh is split into all the matchings of the local rules l.h.s. together with all the inclusion information between these matchings. Any unmatched part of the mesh is lost.
2. *Local rule application* (Fig. 5b ⇒ 5c): each l.h.s. instance is locally replaced by its corresponding r.h.s. The inclusion information is also updated: each l.h.s. inclusion is replaced by its corresponding r.h.s. inclusion.
3. *Reconstruction* (Fig. 5c ⇒ 5d): the inclusion information are finally used to merge the different r.h.s. giving rise to the transformed mesh.

Fig. 5 illustrates the computation of a global transformation step on a mesh composed of four triangles. Note that Figs. 5b and 5c only show more explicitly for the four-triangles case the inclusion structures already described in Fig. 4 for

the two-triangles case.

This construction of the polyhedral subdivision based on *mutual agreement* has to be compared with the equivalent construction in [20] using *mutual exclusion*. Mutual exclusion prevents the computation of refinement in one global step. The solution of [20] consists of a two-step procedure: a first transformation focuses on the subdivision of all edges and a second transformation splits the obtained hexagons and squares into triangles. Exclusion holds since the l.h.s. of each rule consists of only one element (a 1-cell for the first step, a 2-cell for the second step). Obviously the results of both approaches are the same. However the exclusion based approach suffers from two main drawbacks:

1. A marking is required in the facets subdivision step to identify the nodes generated by the edge subdivision step. For example, in the substitution of an hexagon by four triangles, the three newly created edges surrounding the central triangle are incident to new nodes.
2. The intermediate cell complex contains hexagons and squares and then is not a regular triangular mesh. An implementation of this approach requires the use of a data structure allowing the representation of such a complex. This is beyond the ability of data structures classically used in geometric modeling which strongly rely on the triangular nature of the meshes.

5 Global Transformations

In this section, we begin by formalizing the constructions described in the example using categorical concepts. Then, we compare the double-pushout (DPO) approach and global transformations, showing in particular some correspondence in the particular case where some conservation rules are specified. We also compare global transformations to the Amalgamation Theorem. The similarity is explained, and some differences are identified.

5.1 Categorical Characterization

Here, we want to give a formal specification of the objects and constructions presented in Section 4. A *transformation rule* ρ is a pair of two cell complexes, the l.h.s. and the r.h.s. A set of transformation rules gives rise to a function $R_0 : L_0 \to \mathrm{Acc}$, where $L_0 \subset \mathrm{Acc}$ denotes the set of l.h.s., that maps each l.h.s. to its corresponding r.h.s. An *inclusion rule* between two transformation rules ρ and ρ' associates an inclusion of the l.h.s. of ρ' in the l.h.s. of ρ with an inclusion of the r.h.s. of ρ' in the r.h.s. of ρ. Let us denote L_1 the set of all inclusions between any pair of l.h.s. of L_0, the set of inclusion rules can be interpreted as a function R_1 mapping any element of L_1 to its corresponding inclusion in Acc. A set of inclusion rules is said to be *complete* if the associated function is total. A set of inclusion rules is said to be *consistent* if all the inclusions agree on composition. In that sense, the set of inclusion rule given in Fig. 3b for the mesh refinement

example is complete and consistent[4]. It is trivial to see that these two properties induce the determinism of a global transformation since any case that could be encountered is taken into account by a mutual agreement (completeness) and there is no contradiction between these agreements (consistency).

This situation can be nicely summarized in terms of categorical concepts. Considering that L_0 and L_1 form a sub-category $\mathbf{L}$ of the category **AccI**, R_0 and R_1 form a functor R from $\mathbf{L}$ to **AccI**. While consistency holds since R, as a functor, respects the morphism composition in **AccI**, completeness is grabbed when $\mathbf{L}$ is a *full* sub-category where all possible inclusions are considered. This leads to the following definition:

Definition 1 (Global Transformation). *A* global transformation T *is given by a tuple* $\langle \mathbf{L}_T, \mathrm{L}_T, \mathrm{R}_T \rangle$ *where* $\mathbf{L}_T$ *is a full subcategory of* **AccI** *corresponding to the l.h.s. with* $\mathrm{L}_T : \mathbf{L}_T \to$ **AccI** *its associated inclusion functor and* $\mathrm{R}_T : \mathbf{L}_T \to$ **AccI** *is a functor from* $\mathbf{L}_T$ *to* **AccI**.

We now formalize how a global transformation T is applied on a cell complex $\mathcal{K}$ to build the associated result $T\mathcal{K}$. Following the decomposition presented in Section 4.2, we get:

1. *Pattern matching*: all the matchings of the l.h.s. of T in $\mathcal{K}$ and the way they are included one in each other exactly correspond to the objects and morphisms of the comma category $\mathrm{L}_T/\mathcal{K}$. Objects of $\mathrm{L}_T/\mathcal{K}$ are pairs $\langle l \in \mathbf{L}_T, i : \mathrm{L}_T l \to \mathcal{K} \rangle$ where l is a l.h.s. and i an inclusion of this l.h.s. in $\mathcal{K}$. Morphisms of the comma category are inclusions *between* the matchings, *i.e.*, a morphism from a matching $\langle l, i \rangle$ to a matching $\langle l', i' \rangle$ is an inclusion $j : l \to l'$ in $\mathbf{L}$ such that $i = i' \circ \mathrm{L}_T j$.
2. *Local rule application*: in order to get the r.h.s. corresponding to each matching, we first use the projection functor $\mathrm{Proj}_{\mathrm{L}_T/\mathcal{K}} : \mathrm{L}_T/\mathcal{K} \to \mathbf{L}_T$ defined on matchings as $\mathrm{Proj}_{\mathrm{L}_T/\mathcal{K}} = \langle l, i \rangle \mapsto l$, and on matching inclusions as $\mathrm{Proj}_{\mathrm{L}_T/\mathcal{K}} = j \mapsto j$. The functor R_T then maps each l.h.s. to the corresponding r.h.s., and each l.h.s. inclusion to the corresponding r.h.s. inclusion. So the result of this step is the compound functor $\mathrm{R}_T \circ \mathrm{Proj}_{\mathrm{L}_T/\mathcal{K}}$.
3. *Reconstruction*: all the r.h.s. instances are finally glued together w.r.t. $\mathrm{R}_T \circ \mathrm{Proj}_{\mathrm{L}_T/\mathcal{K}}$. The resulting cell complex $T\mathcal{K}$ could be obtained as the colimit of this functor. However, colimits are only guaranteed[5] in **AccM** since the universality may require a non-inclusion morphism to hold. So we use the forgetful functor U to pass from **AccI** to **AccM**.

[4] Only a subset of the inclusion rules is drawn but all other rules can be retrieved using composition of inclusion.

[5] A way to show that all the colimits exist consists in exhibiting an initial object and all pushouts. Here, the initial object is simply the empty cell complex. The pushout of a span of cell complexes can be obtained from the pushout in **Set** of their cells sets by adding the unique possible dimension function and the smallest possible strict partial order relation, which happens to necessarily exist thanks to the dimensions.

A global transformation application is summarized as follows:

Definition 2 (Application of Global Transformation). *Given a cell complex $\mathcal{K}$ and a global transformation T, the result $T\mathcal{K}$ of the application of T on $\mathcal{K}$ is the cell complex $T\mathcal{K} = \mathrm{Colim}(\mathrm{U} \circ \mathrm{R}_T \circ \mathrm{Proj}_{\mathrm{L}_T/\mathcal{K}})$.*

Note that since we consider all the inclusions between the l.h.s. graphs, we also consider their automorphisms, *i.e.*, their symmetries. This means that, in our previous examples, each matching is *really matched* as many times as it has symmetries. The final result remains correct because all the results of these symmetric matchings are glued together appropriately due to the functorial nature of L_T and R_T. Considering these automorphisms is in fact meaningful as they allow to prevent incoherent specifications that intuitively lead to symmetry breaking. Unfortunately, the size of this paper does not allow us to enter into a more detailed discussion about these features.

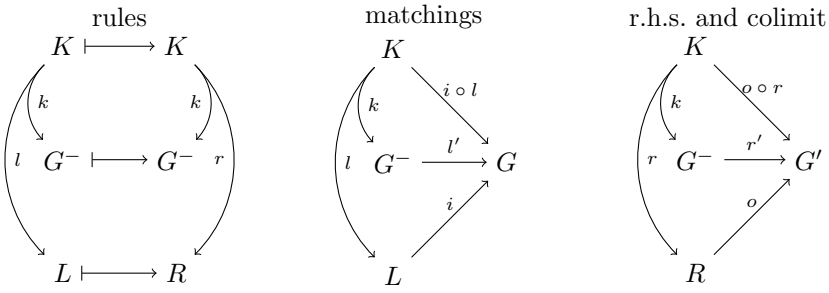
5.2 Double-Pushouts and Global Transformations

One of the consequences of our construction of the transformation result is that all parts of the original cell complex which are not matched are not conserved by any means. This is why we do not consider any morphism relating l.h.s. and r.h.s. If something has to be kept, this must be specified explicitly as we did for the not-refined triangles in Section 4. This is in plain contrast with the DPO approach where the default behavior is to conserve unmatched parts of the graph and deletion has to be specified explicitly.

However, if a global transformation states that some patterns have to be conserved, then it is possible to see the elements of the DPO occurring in a different light. This relation appears because conservation rules have identical l.h.s. and r.h.s. So despite the fact that all considered inclusions are either strictly between l.h.s. or strictly between r.h.s., these conservation rule allows to think inclusions linking both l.h.s. and r.h.s. worlds. To express this a bit more visually, let us consider a derivation $G \Rightarrow G'$ via a production $p = L \leftarrow K \rightarrow R$ based on an inclusion $i : L \rightarrow G$ and try to lay it out in a global transformation application way. The elements of the derivation are given in the following diagram.

$$\begin{array}{ccccc} L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ {\scriptstyle i}\downarrow & & {\scriptstyle k}\downarrow & & {\scriptstyle o}\downarrow \\ G & \xleftarrow{l'} & G^- & \xrightarrow{r'} & G' \end{array}$$

In the global transformation framework, G^- is a compound of many conservation rules, K is a compound of many small conservation rules included in both the conserved and the modified parts, and L and R are all the l.h.s. and r.h.s. of the transformation rules. The result G' is obtained as the colimit of K, G^- and R, which is precisely a pushout. This gives the following layout:

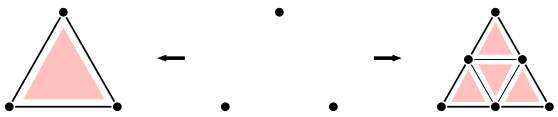


Notice that no notion of pushout complement is required in our construction. The existence of a pushout complement for a production $L \leftarrow K \rightarrow R$ based on a morphism $m : L \rightarrow \mathcal{K}$ imposes that the parts of L which are not in K should not be incident to any cell outside of m for a derivation to be feasible. Contrary to DPOs, all matchings are necessarily applicable in our setting. This gives more flexibility in the design of transformation rules and allows to work exclusively with inclusions. All of this are consequences of the fact that we are in the restricted case where nothing from an input cell complex is considered as conserved.

5.3 Amalgamation and Global Transformations

There is a strong relation between the proposed formalism and the amalgamation of productions available in the DPO approach. Indeed, both consider rules and a notion of inclusions between rules. Moreover, amalgamation of two productions via a sub-production is a pushout of productions and it is well-known that a colimit can be viewed as being mainly a compound of pushouts. If we consider all the matchings of a set of productions and if we have enough sub-productions to handle all possible overlaps, it is clearly possible to synchronize all the productions into one production that operates in a single step as required.

However there are some differences. We have already discussed the default behavior on the unmatched part for both approaches: amalgamation conserves it while global transformation drops it. Assuming that there is no unmatched part, a strong difference remains because of applicability. Indeed, in the DPO approach the amalgamation theorem states that anytime an amalgamation of productions can be applied, each production can be applied individually, with the restriction imposed by the existence of a pushout complement explained earlier. Let us consider the following production that is a straightforward translation of rule ρ_1 of Fig 3a:



Since the three 1-cells in L are not part of K, the rule can only be used on triangles whose edges are incident to no other 0-cells nor 2-cells, which is not

the common case in a triangular mesh. Indeed, in mesh refinement, every cells (except the 0-cells) have to be transformed and the main part of them are incident to each other. This makes impossible in general the design of productions isolating the cells to be transformed, except by considering the whole black region in one global production. As a consequence, there is no simple way to use amalgamation alone to build a single global production from many local ones.

Again, an important property of our approach is that there is no applicability condition over the cell complex to transform. The only constraint comes from the completeness and the consistency of the set of inclusion rules which ensure to get a well defined and unique result.

6 Conclusion and Future Work

In this article, we have presented the Global Graph Transformations, an original categorical framework to deal with deterministic graph transformation with a maximal application strategy. Global transformations are based on the notion of *mutual agreement* (as opposed to *mutual exclusion*) which allows overlapping matchings to agree on the transformation of the shared part. Analogously to amalgamation of productions, mutual agreement is realized by considering additional rules specifying the behavior of the intersection parts. Global transformations are useful for expressing the global rewriting of a graph where no interface is explicitly conserved between the l.h.s. and the r.h.s. Taking a point of view different from substitution processes, a global transformation is a way to construct a set of constraints: local transformation constraints imply that for each l.h.s. matching the corresponding r.h.s. has to appear in the result; inclusion constraints state how r.h.s. have to be glued from l.h.s. inclusions. The resulting graph corresponds to the solution of these constraints.

In this paper, the formalization of global transformations relies on the cell complexes category. We made this choice for bridging with other related works. Firstly, our running example about mesh refinement is definitively more clear without any encoding in usual graphs. Secondly, this approach is in line with the MGS programming language based on the rewriting of topological collections [20]. In this context, labeled cell complexes are reminiscent of the MGS topological collections and global transformations correspond to a new rule application strategy. We plan to integrate this strategy in the current implementation of the language. Finally, *causal graphs dynamics* proposed in [1] have been recently extended to *combinatorial manifolds* [2], a specific class of cell complexes. Similarly to these works, we are interested in understanding the notion of locality and causality in global transformations and to relate these notions to a kind of topological continuity of the transformation. In this context, using cell complexes seems more natural since they have been the subject of many developments in algebraic and digital topology.

From a pure algebraic point of view, the use of cell complexes is a detail. For example, the global transformation framework can be moved seamlessly from cell complexes to graphs. One can then ask what kind of general theory supports

global transformations in the same way that the essential properties of graphs regarding DPOs have been identified leading to several axiomatic frameworks (*e.g.*, adhesive [9] and High-Level Replacement [13] categories). There also seems to be interesting possibilities in integrating the notion of nested application condition in the context of global transformations too, to specify for example that a sub-rule is only considered as the intersection of some super-rules and not others. All these issues are part of future work.

Acknowledgments

This research is partially supported by the French ANR grants "SynBioTIC" ANR-10-BLAN-0307 and "TARMAC" ANR-12-BS02-0007, and by the University Paris-Est Créteil.

References

1. Arrighi, P., Dowek, G.: Causal graph dynamics. CoRR abs/1202.1098 (2012)
2. Arrighi, P., Martiel, S., Wang, Z.: Causal dynamics of discrete surfaces. In: DCM 2013, Buenos Aires, Argentina, 26 August 2013. pp. 30–40 (2014)
3. Boehm, P., Fonio, H.R., Habel, A.: Amalgamation of graph transformations: A synchronization mechanism. Journal of Computer and System Sciences 34(2–3), 377 – 408 (1987)
4. Ehrig, H., Rosen, B.K.: Parallelism and concurrency of graph manipulations. Theoretical Computer Science 11(3), 247 – 275 (1980)
5. Giavitto, J.L., Michel, O., Cohen, J., Spicher, A.: Computation in space and space in computation. In: UPP'04. LNCS, vol. 3566, pp. 137–152. Spinger, Le Mont Saint-Michel (Sep 2005)
6. Golas, U., Habel, A., Ehrig, H.: Multi-amalgamation of rules with application conditions in m-adhesive categories. Mathematical Structures in Computer Science 24(4), 1 – 68 (2014)
7. Janssens, D., Rozenberg, G., Verraedt, R.: On sequential and parallel node-rewriting graph grammars. Computer Graphics and Image Processing 18(3), 279–304 (1982)
8. Kovalevsky, V.A.: Geometry of Locally Finite Spaces. Editing House Dr. Baerbel Kovalevsky (2008)
9. Lack, S., Sobociński, P.: Adhesive categories. In: FOSSACS 2004, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings. pp. 273–288. Springer (2004)
10. Lane, S.M.: Categories for the working mathematician, vol. 5. Springer Science & Business Media (1978)
11. Loop, T.L.: Smooth subdivision surfaces based on triangle. Master's thesis, University of Utah (Aug 1987)
12. Munkres, J.: Elements of Algebraic Topology. Addison-Wesley (1984)
13. Padberg, J.: Survey of high-level replacement systems. Tech. rep. (1993)
14. Paszyńska, A., Grabska, E., Paszyński, M.: A graph grammar model of the hp adaptive three dimensional finite element method. part i. Fundamenta Informaticae 114(2), 149–182 (2012)

15. Paun, G.: Computing with membranes. Journal of Computer and System Sciences 1(61), 108–143 (2000)
16. Prusinkiewicz, P., Lindenmayer, A., Hanan, J.S., et al.: The Algorithmic Beauty of Plants. Springer-Verlag (1990)
17. Raoult, J.C., Voisin, F.: Set-theoretic graph rewriting. Tech. Rep. RR-1665, INRIA (April 1992)
18. Rozenberg, G., Ehrig, H.: Handbook of graph grammars and computing by graph transformation, vol. 1. World scientific Singapore (1999)
19. Rozenberg, G., Salomaa, A.: Lindenmayer Systems. Springer, Berlin (1992)
20. Spicher, A., Michel, O., Giavitto, J.L.: Declarative mesh subdivision using topological rewriting in MGS. In: ICGT 2010. LNCS, vol. 6372, pp. 298–313 (2010)

Parallel evaluation of interaction nets: some observations and examples
(Work-in-progress)

Ian Mackie and Shinya Sato

Abstract. Interaction nets are a particular kind of graph rewriting system that have many properties that make them useful for capturing sharing and parallelism. There have been a number of research efforts towards implementing interaction nets in parallel, and these have focused on the implementation technologies. In this paper we investigate a related question: when is an interaction net system suitable for parallel evaluation? We observe that some nets are cannot benefit from parallelism (they are sequential) and some have the potential to be evaluated in a highly parallel way. This first investigation aims to highlight a number of issues, by presenting experimental evidence for a number of case studies. We hope this can be used to help pave the way to a wider use of this technology for parallel evaluation.

1 Introduction

Interaction nets are a model of computation based on a restricted form of graph rewriting: the rewrite rules must be between two nodes on the left-hand side, be local (not change any part of graph other than the two nodes), and there must be at most one rule for each pair of nodes. These constraints have no impact on the expressive power of interaction nets (they are Turing complete), but they offer a very useful feature: they are confluent by construction. Taken with the locality constraint they lend themselves to parallel evaluation: all rewrite rules that can apply can be rewritten in one parallel step.

The question that we propose in this paper is: when is a particular interaction net system well suited for parallel evaluation. More precisely, are some interaction nets "more parallel" than others? A question that naturally follows from this is can we transform a net so that it is more suited for parallel evaluation. Once we have understood this, we can also ask the reverse question: can a net be made sequential? The purpose of this paper is to make a start to investigate these questions, and we begin with an empirical study of interaction systems to identify when they are suitable for parallel evaluation or not.

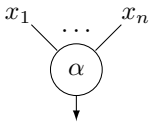
We take a number of typical examples (some common ones from the literature together with some new ones we made up for this paper) to see if they benefit from parallel evaluation. In addition, we make some observations about how programs can be transformed so that parallelism is more useful. Using these examples, we give some heuristics for getting more parallelism out of an interaction net system.

Related work. There have been a number of studies for the parallel implementation of interaction nets: Pinto [6] and Jiresch [3] are two examples. In these works it is the implementation of a given net that has been the focus. Here we are interested in knowing if a net is well suited for parallel evaluation or not.

Structure. In the next section we recall the definition of interaction nets, and describe the notion of parallel evaluation that we are interested in. Through examples we motivate the ideas behind this work. In Section 3 we give a few small case studies to show how parallelism can have a significant impact on the evaluation of a net. In Section 4 we give a short discussion and conclude in Section 5.

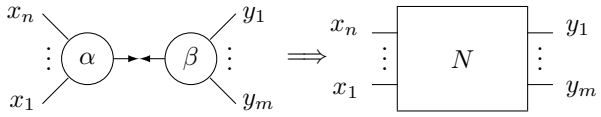
2 Background and Motivation

In the graphical rewriting system of interaction nets [4], we have a set Σ of *symbols*, which are names of the nodes in our diagrams. Each symbol has an arity ar that determines the number of *auxiliary ports* that the node has. If $ar(\alpha) = n$ for $\alpha \in \Sigma$, then α has $n + 1$ *ports:* n auxiliary ports and a distinguished one called the *principal port*.



Nodes are drawn variably as circles, triangles or squares. A *net* built on Σ is an undirected graph with nodes at the vertices. The edges of the net connect nodes together at the ports such that there is only one edge at every port. A port which is not connected is called a *free port*.

Two nodes $(\alpha, \beta) \in \Sigma \times \Sigma$ connected via their principal ports form an *active pair*, which is the interaction nets analogue of a redex. A rule $((\alpha, \beta) \Longrightarrow N)$ replaces the pair (α, β) by the net N. All the free ports are preserved during reduction, and there is at most one rule for each pair of agents. The following diagram illustrates the idea, where N is any net built from Σ.



The most powerful property of this system is that it is one-step confluent: the order of rewriting is not important, and all sequences of rewrites are of the same length (in fact they are permutations). This has practical consequences: the diagrammatic transformations can be applied in any order, or even in parallel, to give the correct answer. It is the latter feature that we develop in this paper.

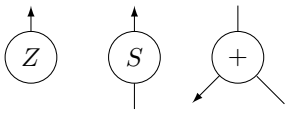
We define some notions of nets and evaluation. A net is called *sequential* if there is at most one active pair that can be reduced at each step. We say that a

net is evaluated sequentially if one active pair is reduced at each step. For our notion of parallel evaluation, we require that all active pairs in a net are reduced simultaneously, and then any redexes that were created are evaluated at the next step. We do not bound the number of active pairs that can be reduced in parallel. We remark that the number of parallel steps will always be less than or equal to the number of sequential steps (for a sequential net, the number of steps is the same for sequential and parallel evaluation).

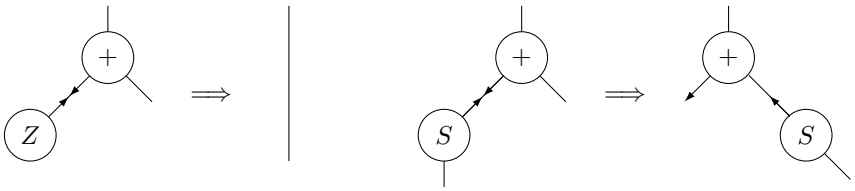
As an example, consider unary numbers with addition. We represent the following term rewriting system

```
add(Z,y)    = y
add(S(x),y) = add(x,S(y))
```

as a system of nets with agents Z, S, $+$:



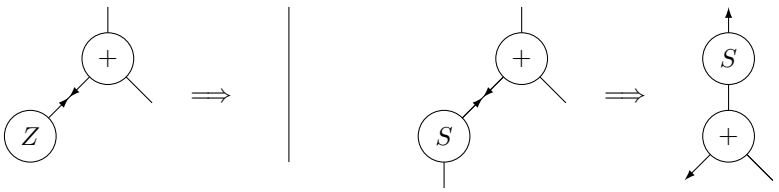
together with two rewrite rules:



We observe that addition of two numbers is sequential: at any time there is just one active pair, and reducing this active pair creates one more active pair, and so on. In terms of cost, reducing `add(n,m)` requires $n + 1$ interactions. If we consider the net corresponding to the term $add(add(m, n), p)$, then the system is sequential, and the costs are now $2m+n+2$. Using associativity of addition, the situation changes significantly. The net corresponding to $add(m, add(n, p))$ has sequential cost $m+1+n+1 = m+n+2$, and parallel cost $max(m+1, n+1)$. This is significantly more efficient sequentially, and moreover is able to benefit from parallel evaluation. The example becomes even more interesting if we change the system to an alternative version of addition:

```
add(Z,y)    = y
add(S(x),y) = S(add(x,y))
```

The two interaction rules are now:



Unlike the previous system, the term $add(add(m,n),p)$ already has scope for parallelism. The sequential cost is now $2m+n+2$ and the parallel cost is $m+n+2$. But again, if we use associativity then we can do even better and achieve sequential cost $m+n+2$ and parallel cost $max(m+1,n+1)$ for the term $add(m,add(n,p))$.

These examples illustrate that some nets are sequential; some nets can use properties of the system (in this case associativity of addition) to get better sequential and parallel behaviours; and some systems can have modified rules that are more efficient, and also more appropriate to exploit parallelism. The next section gives examples where there is scope for parallelism in nets.

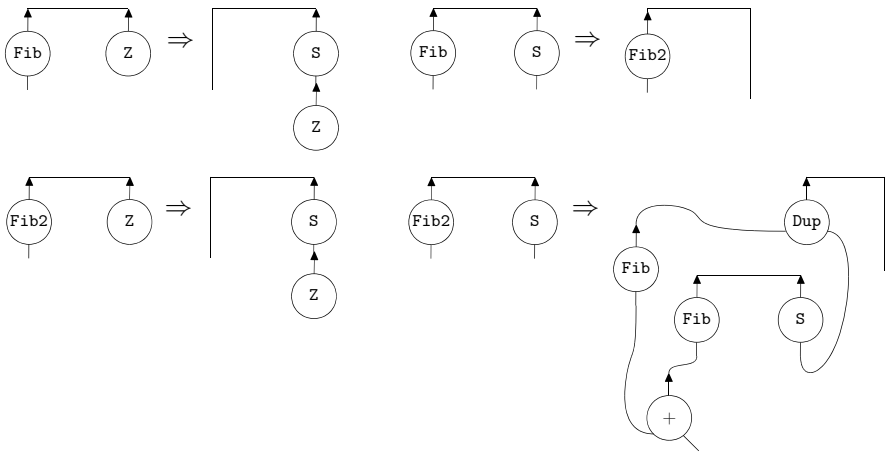
3 Case studies

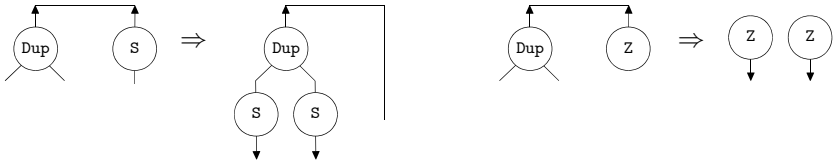
The previous arithmetic example demonstrates that some systems are more useful than others for parallel evaluation. In this section we give some empirical case studies for a number of different systems to show that when a suitable system can be found, the parallel evaluation gives significantly better results than sequential evaluation.

Fibonacci. The Fibonacci function is a good example where many recursive calls generate a lot of possibilities for parallel evaluation. We build the interaction net system that corresponds to the term rewriting system:

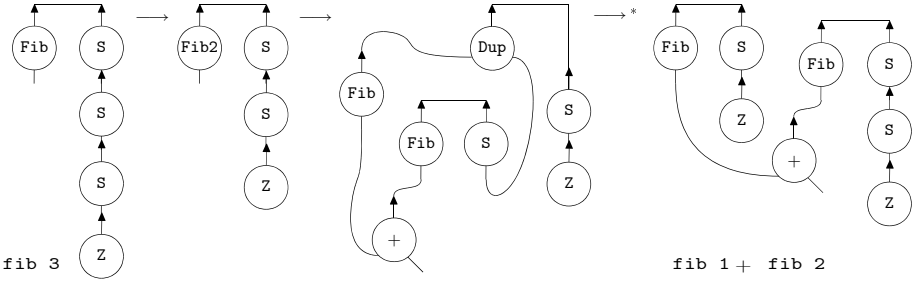
```
fib 0 = fib 1 = 1
fib n = fib(n-1) + fib(n-2)
```

Using a direct encoding of this system together with addition defined previously, we can obtain an interaction system:



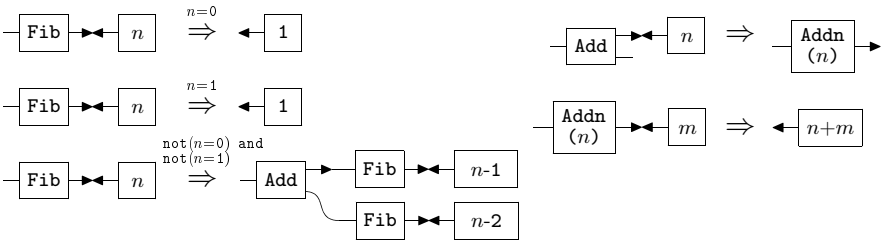


The following is an example of rewriting:



With respect to the two versions of the addition operation introduced in Section 2, we call the former a *batch* operation, which returns the computational result after finishing processing all of the given data, and the latter a *streaming* operation, which computes one (or a small number of) elements of the given data and returns partial parts of the computational result immediately. The graphs in Figure 1 show the number of interactions in each version, where we plot sequential steps against parallel steps to indicate the rate of growth of each one. Both graphs demonstrate that the sequential computation is exponential, while the parallel one is quadratic. We remark that, in the parallel execution, the numbers of steps with the streaming operation are less than a half of the numbers with the batch operation. This result is illustrated in the third graph in the figure.

By allowing attributes as labels of agents, we can include integer numbers in agents. In addition, we can use conditional rewritings, preserving the one-step confluence, when these conditions on attributes are disjoint. In this case, the system of the Fibonacci function is written as follows:



There is very little difference between the load balances of `fib` $(\mathtt{n}-\mathtt{1})$ and `fib` $(\mathtt{n}-\mathtt{2})$, and thus this system gives the following graph, demonstrating that the growth rate for parallel computation is linear, while the sequential rate is exponential:

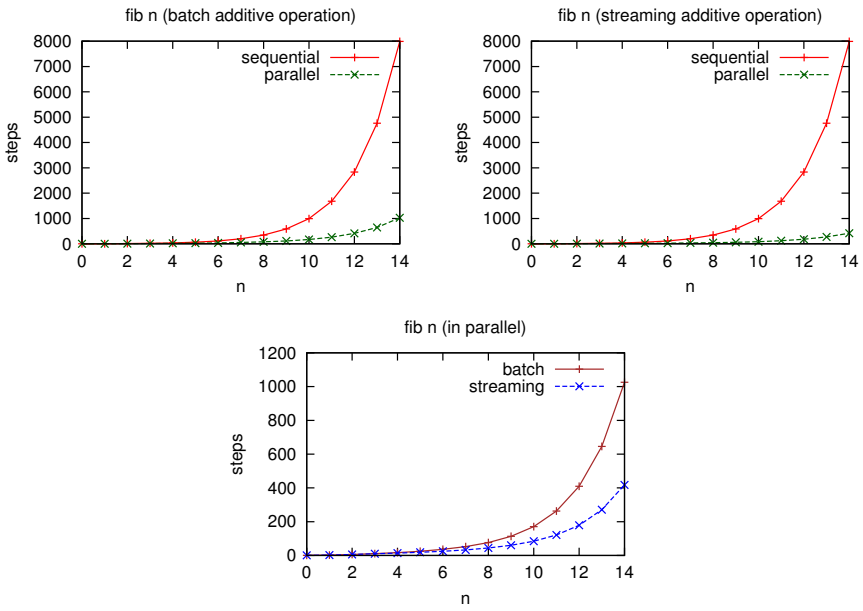
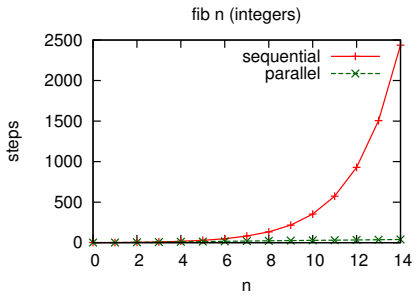
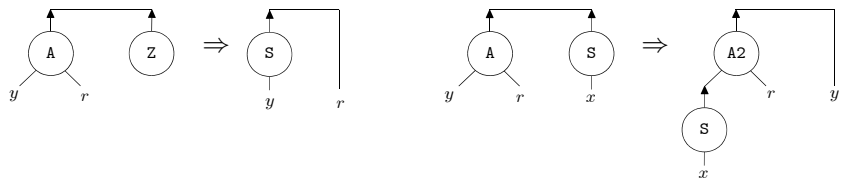
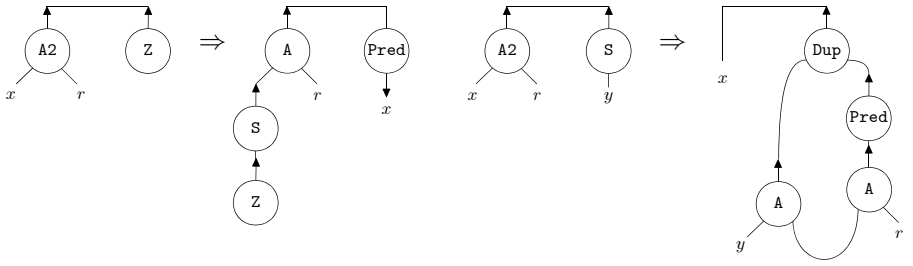


Fig. 1. Comparing batch and streaming operations

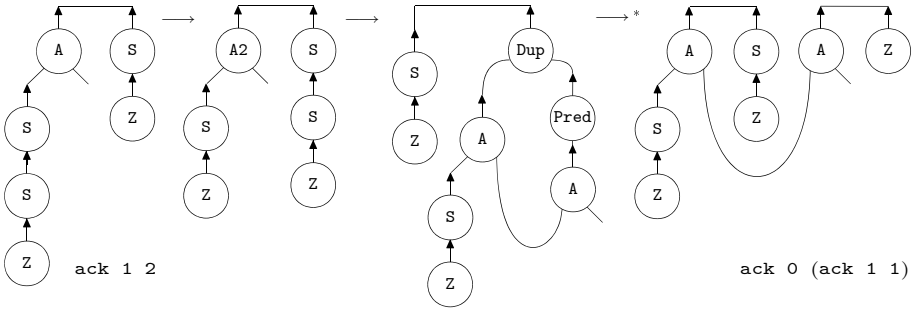


Ackermann. The Ackermann function is defined by three cases: `ack 0 n = n+1`, `ack m 0 = ack (m-1) 1`, and `ack m n = ack (m-1) (ack m (n-1))`. We can build the interaction net system on the unary natural numbers that corresponds to the term rewriting system as follows:





where the agent `Dup` duplicates `S` and `Z` agents. The following is an example of rewriting:



When we use numbers as attributes, the system can be written as:

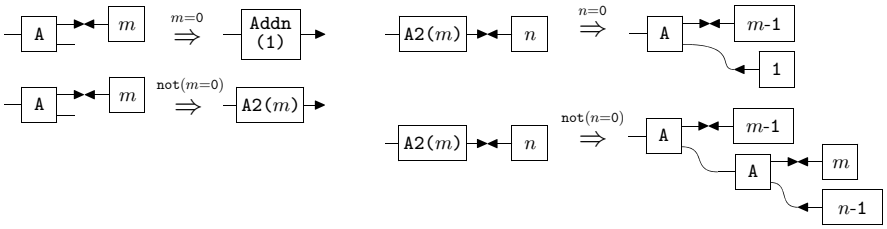
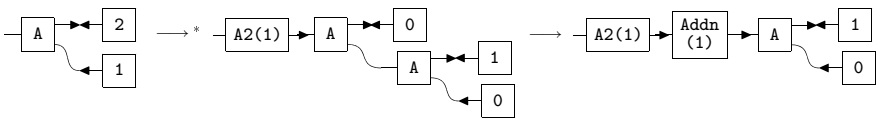


Figure 2 shows the number of interactions in the cases of (a) unary natural numbers and (b) integer numbers, where we plot sequential steps against parallel steps to indicate the rate of growth of each one. Unfortunately, in Figure 2 (b), there is no significant difference in the sequential and the parallel execution, and thus there is no possibility of the improvement by the parallel execution. This is because the `Addn` agent works as the batch operation, thus it waits for part of the result. For instance, after the last step in the following the computation step `ack 2 1`, the `Addn(1)` agent, which is the result of `ack 0 (ack 1 0)`, waits the computational result of `ack 1 0`. However, the computation of `A2` should proceed because the result of the `Addn(1)` will be more than 0.



(a) unary natural numbers (b) integers

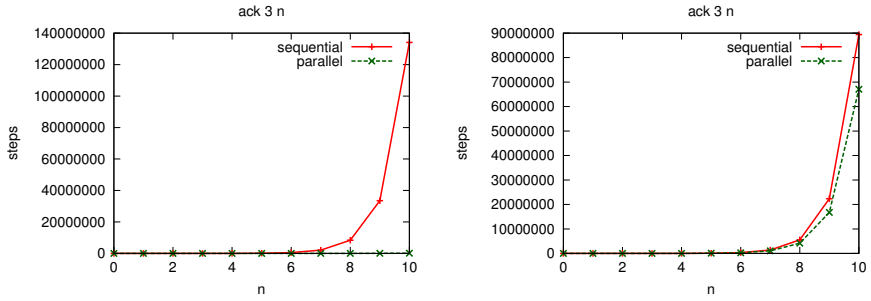
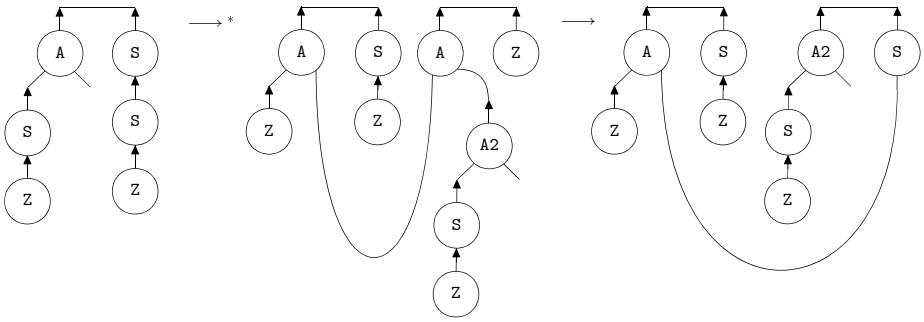
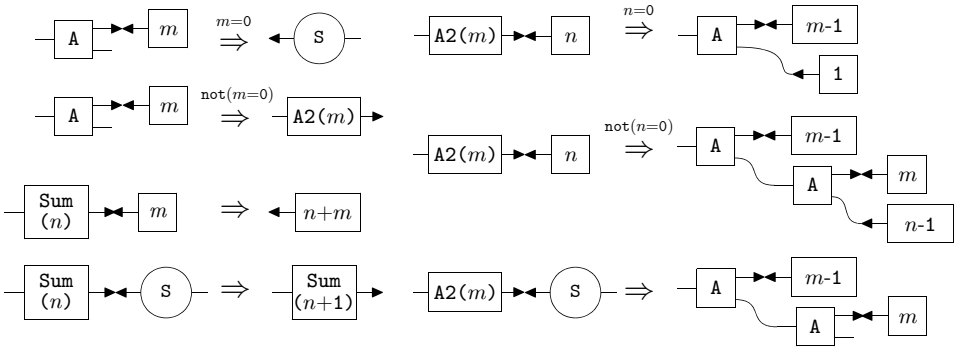


Fig. 2. Benchmarks of the execution of Ackermann function in sequential and parallel

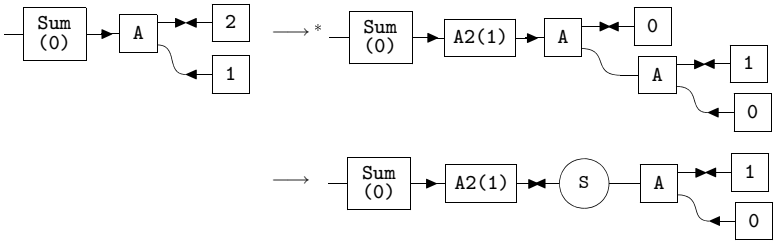
On the other hand, in the case of the computation on unary natural numbers, the `A2` interacts with the streaming result of `ack 0` (`ack 1 0`):



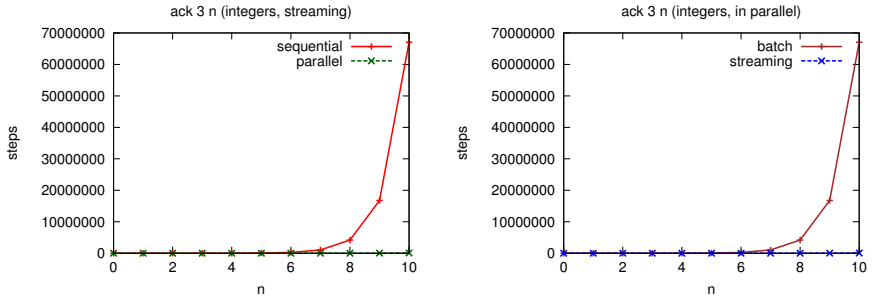
Here, borrowing the `S` agent to denote numbers greater than 0, we change the rules, especially in the case of `ack 0 n`, `Addn` into `S` as follows:



Thanks to the introduction of the `S` agent, `A2` can be processed without waiting for the result of `ack 1 0`. This therefore gives a streaming operation:



In addition, the benchmark graph shows that the improved system is more efficient and more appropriate to exploit parallelism:

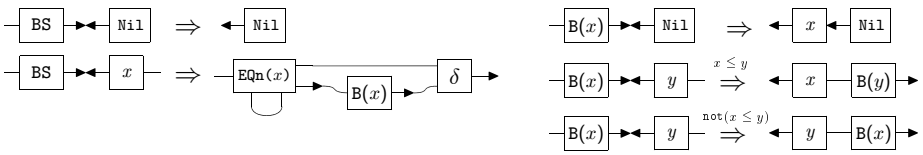


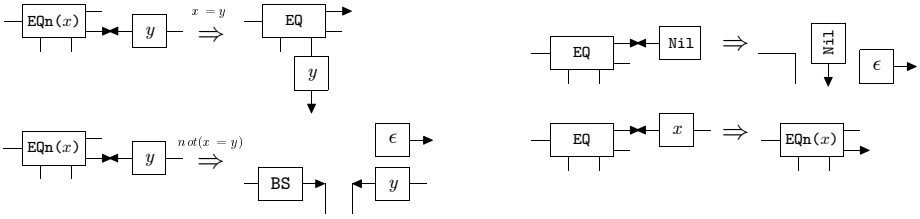
To summarise this section, a system can exploit parallelism by changing some batch operations into streaming ones. We leave as future work the criteria to determine when this transformation can benefit from parallelism.

Sorting. Bubble sort is a very simple sorting algorithm that can benefit from parallel evaluation in interaction nets. One version of this algorithm, written in Standard ML [5], is as follows:

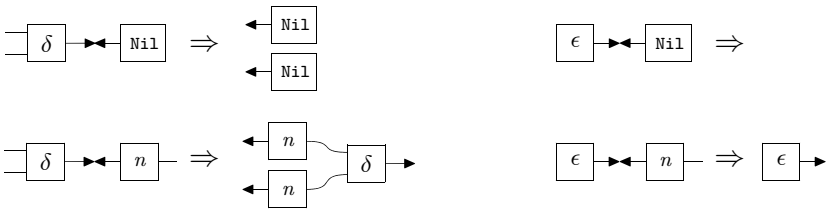
```
fun bsortsub (x::x2::xs) =
  if x > x2 then x2::(bsortsub (x::xs)) else x::(bsortsub(x2::xs))
| bsortsub x = x

fun bsort t =
  let val s = bsortsub t
   in if t=s then s else bsort s
  end;
```

Using a direct encoding of this program, we obtain the interaction system:

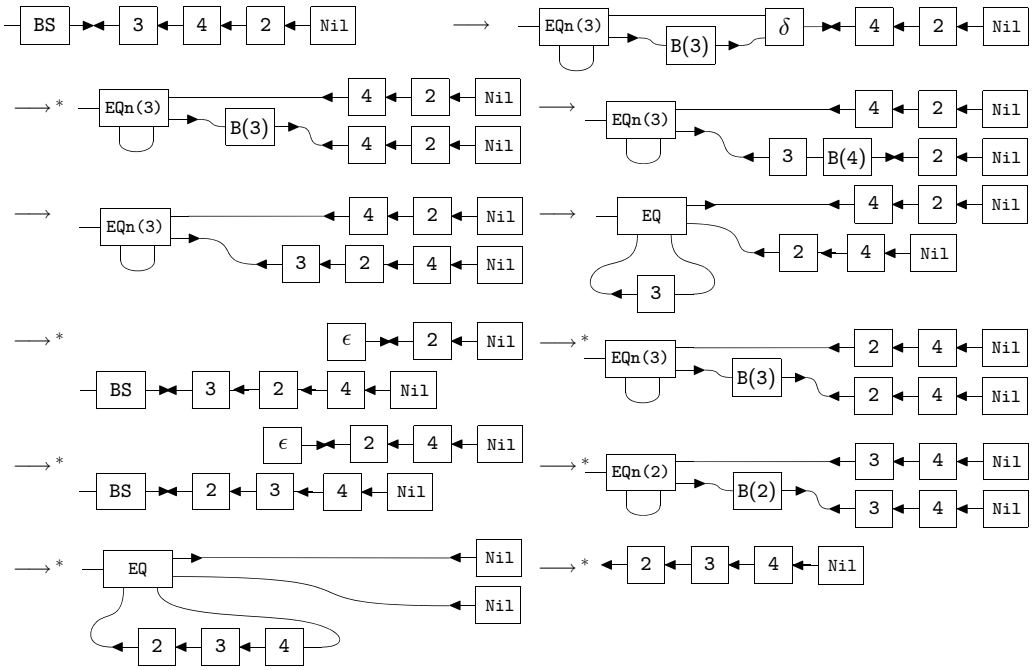




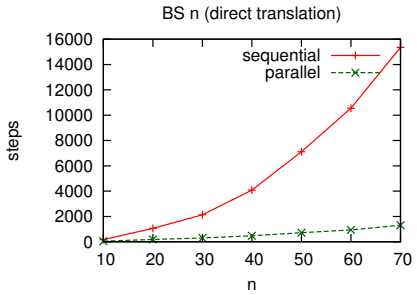
where the δ and ϵ agents are defined as a duplicator and an eraser:



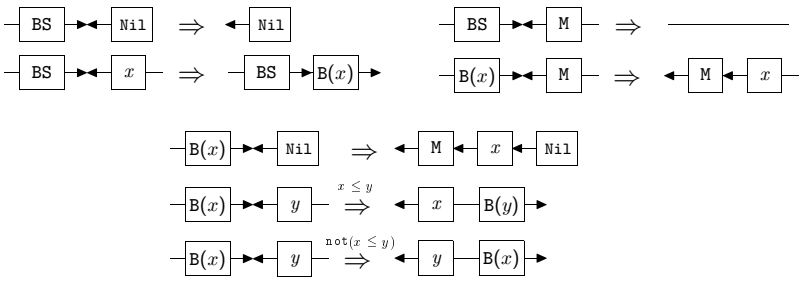
For instance, a list $[3, 4, 2]$ is sorted as follows:



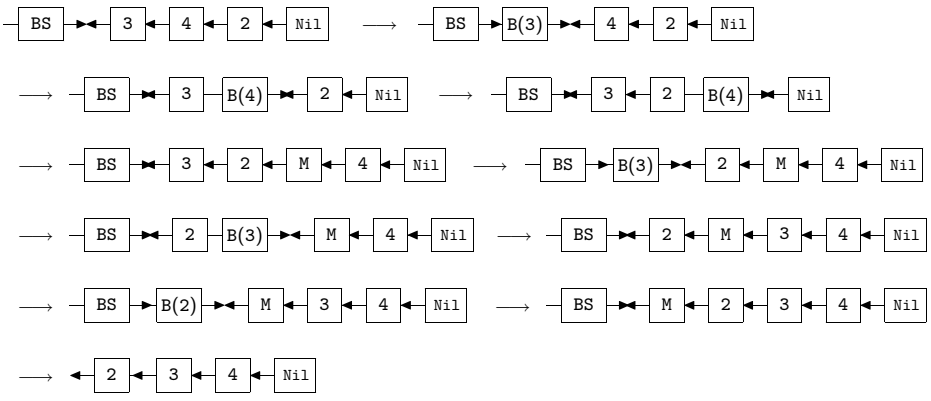
This system shows that parallel bubble sorting is linear, whereas sequential evaluation is quadratic, as indicated in the graph below.



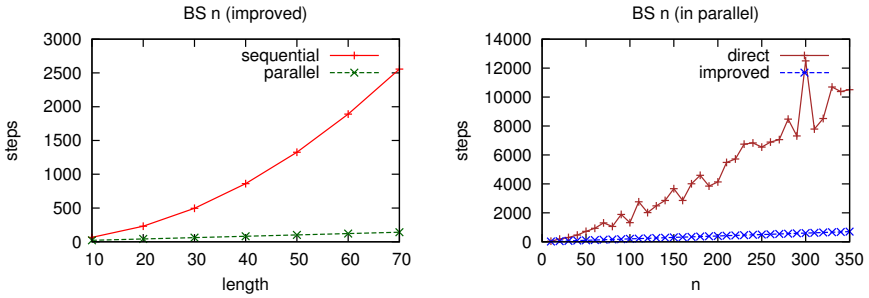
However, it contains the equality test operation by `EQ` and `EQn` to check whether the sorted list is the same as the given list. In comparison to the typical functional programming languages, interaction nets require copying and erasing of lists for the test that can cause inefficient computation. Moreover, the sorting process is applied to the sorted list by `B` again and again. Taking into account that the `B` moves the maximum number in the given unsorted list into the head of the sorted list, we can obtain a more efficient system:



For instance, a list $[3, 4, 2]$ is sorted as follows:



The system reduces the number of computational steps significantly, and gives the best expected behaviour as follows:



Summary/discussion. All these examples show the scope for harnessing parallelism from an empirical study: some systems do not benefit, whereas others allow quadratic computations be executed in linear parallel complexity. However, these results give a flavour of the potential, and do not necessarily mean that they can be implemented like this in practice.

4 Discussion

In this section we examine the potential of parallelism illustrated by the graphs in Section 3, by using a multi-threaded parallel interpreter of interaction nets, called *Inpla*, implemented with gcc 4.6.3 and the Posix-thread library.

We compare the execution time of Inpla with other evaluators and interpreters. The programs were run on a Linux PC (2.4GHz, Core i7, 16GB) and the execution time was measured using the UNIX `time` command as the average of five executions.

First, in executions of the pure interaction nets, we take INET [1] and amineLight [2] and compare Inpla with those by using programs – Fibonacci function (streaming additive operation) and Ackermann function. Table 1 shows execution time in seconds among interaction nets evaluators. We see that Inpla runs faster than INET since Inpla is a refined version of amineLight, which is the fastest interaction nets evaluator [2]. In the table the subscript of Inpla gives the number of threads in the thread pool, for instance Inpla_2 means that it was executed by using two threads. Generally, since Core i7 processor has four cores, it tends to reach the peak with four execution threads.

Next, we compare Inpla with Standard ML of New Jersey (SML v110.74) [5] and Python (2.7.3) [7] in the extended framework of interaction nets which includes integer numbers and lists. SML is a functional programming language and it has the eager evaluation strategy that is similar to the execution method in interaction nets. Python is a widely-used interpreter, and thus the comparison with Python gives a good indication on efficiency. Here we benchmark the Fibonacci function and the streaming operation versions of Ackermann and the improved version of Bubble Sort algorithm for randomly generated list elements. Table 2 shows that SML computes those arithmetic functions fastest. Inpla uses

agents to represent the functions and integer numbers, and those agents are consumed and reproduced repeatedly during computation. Thus the execution time becomes slower eventually, compared to the execution in SML that performs computation by function calls and managing stacked arguments. In comparison with Python, Inpla computes those functions faster. The sort algorithm is a special case in that interaction nets are efficient to implement these algorithms.

	INET	amLight	Inpla	Inpla_1	Inpla_2	Inpla_3	Inpla_4	Inpla_5
`fib 29`	2.31	2.05	1.29	1.31	1.00	0.93	0.90	0.92
`fib 30`	3.82	3.40	1.74	1.74	1.24	1.13	1.08	1.12
`ack 3 10`	18.26	11.40	4.24	4.42	2.33	1.66	1.36	1.44
`ack 3 11`	66.79	46.30	17.53	18.13	9.47	6.67	5.86	5.83

Table 1. The execution time in seconds on interaction nets evaluators

	SML	Python	Inpla	Inpla_1	Inpla_2	Inpla_3	Inpla_4	Inpla_5
`fib 34`	0.12	2.09	1.67	1.50	0.80	0.70	0.68	0.82
`fib 38`	0.66	16.32	11.39	10.22	5.68	4.47	4.40	4.75
`ack 3 6`	0.03	0.05	0.02	0.03	0.02	0.02	0.02	0.02
`ack 3 9`	0.06	-[1]	0.69	0.72	0.38	0.27	0.24	0.24
`BS 10000`	1.64	6.71	2.11	2.25	1.17	0.87	0.76	0.68
`BS 20000`	8.38	30.35	8.38	8.93	4.57	3.64	2.98	2.49

[1] RuntimeError: maximum recursion depth exceeded

Table 2. The execution time in seconds on interpreters

Next we analyse the results of the parallel execution in Inpla by using graphs in Section 3, which show the trends of steps in parallel execution on the assumption of the unbounded resources. We may write "`parallel`(n)" in the following graphs to make explicit that Inpla_n is used for the experiment.

Fibonacci function. Figure 3 shows the execution time of each program for Fibonacci function by using Inpla. We see that each the sequential execution is exponential as shown in the graphs on the assumption of the unbounded resources. The increase rate of execution time in the parallel execution by Inpla gradually becomes close to, according to increasing the number of threads, the trends of the parallel computation in the graphs on the assumption.

We note that, in the computation of unary natural numbers, the execution of the streaming version is slower than the batch version as shown in the graph on the left side in Figure 4. The graph on the right side shows the ratio of steps in the streaming version to steps in the batch version on the assumption of the

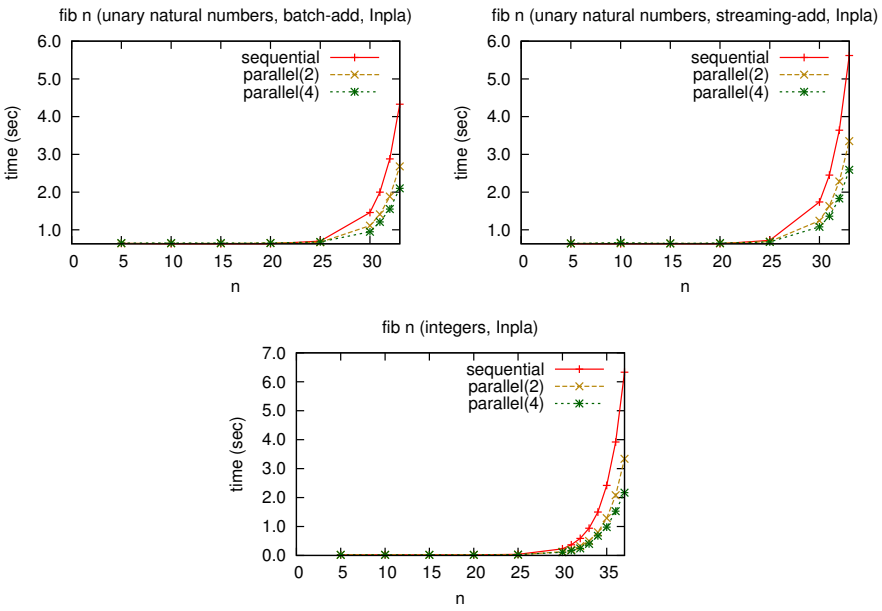


Fig. 3. The execution time of Fibonacci function by Inpla

unbounded resources. The ratio becomes around 0.4 according to increasing n in `ack 3` n. This means that there is a limited benefit of the parallelism, even if we assume unbounded resources. In the real computation, the cost of parallel execution more affects the execution time in comparison to the benefit of the parallelism, and thus the streaming version becomes slower.

Ackermann function. Figure 5 shows the execution time of each program for Ackermann function by using Inpla. We see that, except for the batch operation version, the parallel computation follows well the trends on the assumption of the unbounded resources. On the other hand, the parallel execution of the batch operation version takes quite a long time compared to the streaming version. This is because, in the unbounded resources, not only that there is no significant difference in sequential and parallel execution, but also that there is a cost of parallel execution such as scheduling of threads execution uselessly. These are some of the reasons why the parallel execution does not always have good performance, but are improved in the streaming version.

Bubble sort. Figure 6 shows the execution time of the two programs for Bubble sort using Inpla. As anticipated by the graphs on the assumption of the unbounded resources, we see that the improved version performs best as expected.

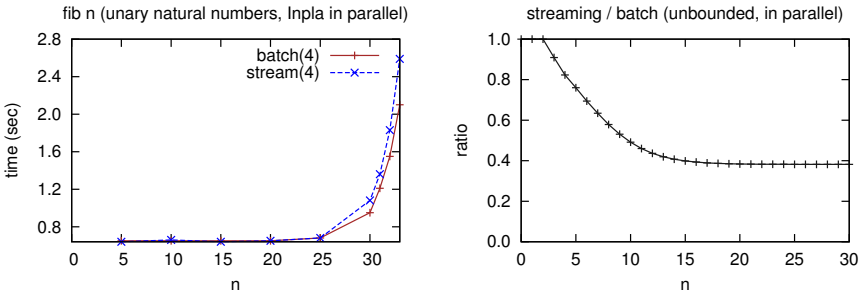


Fig. 4. Comparison between the batch and the streaming addition in parallel execution by Inpla

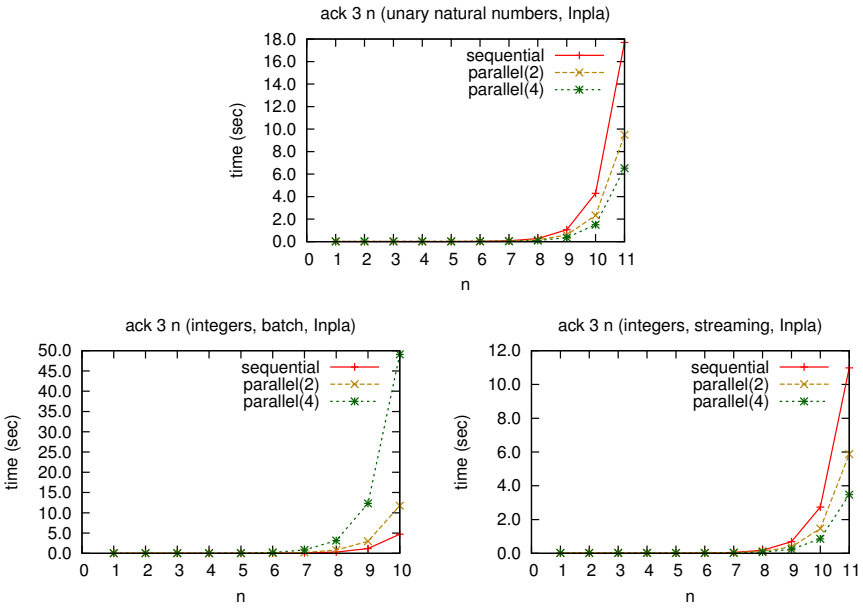


Fig. 5. The execution time of Ackermann function by Inpla

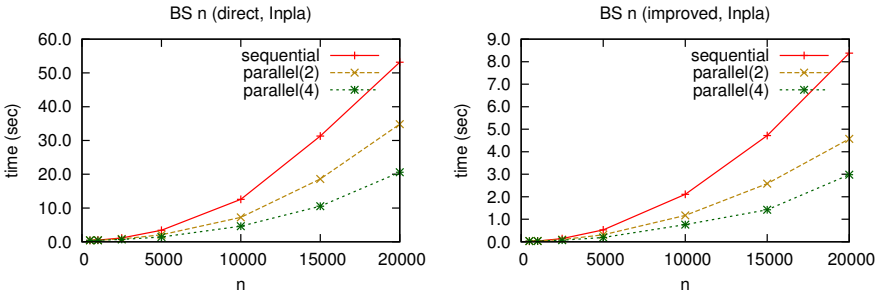


Fig. 6. The execution time of Bubble sort by Inpla

5 Conclusion

Although discussed for many years, we believe that parallel implementations of interaction nets is still a very new area and much needs to be done. In this work we have assumed unbounded resources in terms of the number of processing elements available. This is a reasonable assumption with GPU when many thousands of processing elements are available. We analysed the execution result of the multi-threaded execution by using the investigation result on the assumption, and also showed that, on the one hand, these perform as the best expected, and on the other hand, some of execution results take something away from the investigation results due to an overhead of using parallel technologies as anticipated by the investigation. We hope the ideas in this paper may help in moving this work forward.

References

1. A. Hassan, I. Mackie, and S. Sato. Compilation of interaction nets. *Electr. Notes Theor. Comput. Sci.*, 253(4):73–90, 2009.
2. A. Hassan, I. Mackie, and S. Sato. A lightweight abstract machine for interaction nets. *ECEASST*, 29, 2010.
3. E. Jiresch. Towards a gpu-based implementation of interaction nets. In B. Löwe and G. Winskel, editors, *DCM*, volume 143 of *EPTCS*, pages 41–53, 2014.
4. Y. Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108. ACM Press, 1990.
5. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
6. J. S. Pinto. Sequential and Concurrent Abstract Machines for Interaction Nets. In J. Tiuryn, editor, *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS)*, number 1784 in Lecture Notes in Computer Science, pages 267–282. Springer-Verlag, 2000.
7. G. van Rossum and F. L. Drake. *The Python Language Reference Manual*. Network Theory Ltd., 2011.

Attribution of Graphs by Composition of $\mathcal{M}, \mathcal{N}$-adhesive Categories

Christoph Peuser* and Annegret Habel

Carl von Ossietzky Universität Oldenburg
{peuser,habel}@informatik.uni-oldenburg.de

Abstract. This paper continues the work on $\mathcal{M}, \mathcal{N}$-adhesive categories and shows some important constructions on these categories. We use these constructions for an alternative, short proof for the $\mathcal{M}, \mathcal{N}$-adhesiveness of partially labelled graphs. We further present a new concept of attributed graphs and show that the corresponding category is $\mathcal{M}, \mathcal{N}$-adhesive. As a consequence, we inherit all nice properties for $\mathcal{M}, \mathcal{N}$-adhesive systems such as the Local Church-Rosser Theorem, the Parallelism Theorem, and the Concurrency Theorem for this type of attributed graphs.

Keywords: Graph transformation, attributed graphs, composition, adhesive categories, adhesive systems

1 Introduction

The double-pushout approach to graph transformation, which was invented in the early 1970's, is the best studied framework for graph transformation [Roz97], [EEKR99, EKMR99, EEPT06b]. As applications of graph transformation come with a large variety of graphs and graph-like structures, the double-pushout approach has been generalized to the abstract settings of high-level replacement systems [EHKP91], adhesive categories [LS05], $\mathcal{M}$-adhesive categories [EGH10], $\mathcal{M}, \mathcal{N}$-adhesive categories [HP12], and $\mathcal{W}$-adhesive categories [Gol12]. This paper continues the work of Habel and Plump [HP12] on $\mathcal{M}, \mathcal{N}$-adhesive categories.

In the literature, there are several variants of attribution concepts, e.g. typed attributed graphs in the sense of Ehrig et al. [EEPT06b], attributed graphs in the sense of Plump [Plu09], attributed graphs as a graph with a marked sub-graph in the sense of Kastenberg and Rensink [KR12], separation of the graph structure and their attribution and data in the sense of Golas [Gol12], and attributed structures in the sense of Duval et al. [DEPR14].

Our main aim is to introduce a simple, alternative concept for attributed graphs and attributed graph transformation. Our approach is to define a category

* This work is supported by the German Research Foundation through the Research Training Group (DFG GRK 1765) SCARE (www.scare.uni-oldenburg.de).

AttGraphs of attributed graphs from the well-known category **Graphs** of unlabelled graphs and a category **Att** of attribute collections by a multiset construction and the comma category construction. By closure results for $\mathcal{M},\mathcal{N}$-adhesive categories, we obtain that the category **AttGraphs** is $\mathcal{M},\mathcal{N}$-adhesive. By the results in [HP12], the Local Church-Rosser Theorem, the Parallelism Theorem and the Concurrency Theorem hold for the new type of attributed graphs provided that the HLR$^+$-properties are satisfied.

The paper is structured as follows. In Section 2, we recall the definition of $\mathcal{M},\mathcal{N}$-adhesive categories. In Section 3, we prove some basic composition results and show that constructions for a string and a multiset category are $\mathcal{M},\mathcal{N}$-adhesive for suitable classes $\mathcal{M}$ and $\mathcal{N}$ provided that the underlying category is. As a consequence, the category of partially labelled graphs is $\mathcal{M},\mathcal{N}$-adhesive, as shown in Section 4. In Section 5, we introduce a new concept of attributed graphs - similar to partially labelled graphs - and show that the corresponding category of attributed graphs is $\mathcal{M},\mathcal{N}$-adhesive. In Section 6, we present a precise relationship between $\mathcal{M},\mathcal{N}$-adhesive and $\mathcal{W}$-adhesive categories, in Section 7 some related work, and in Section 8 some concluding remarks.

Note that this paper comes with a long version [PH15] containing the proofs and additional examples.

2 $\mathcal{M},\mathcal{N}$-adhesive Categories

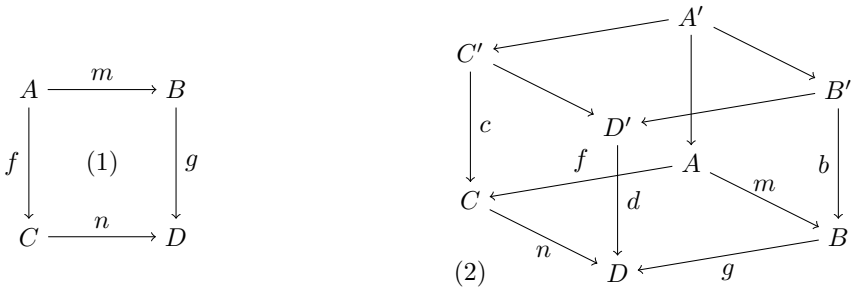
In this section, we recall the definition of $\mathcal{M},\mathcal{N}$-adhesive categories, introduced in [HP12], generalizing the one of $\mathcal{M}$-adhesive categories [EGH10]. We assume that the reader is familiar with the basic concepts of category theory; standard references are [EEPT06b, Awo10].

Definition 1 ($\mathcal{M},\mathcal{N}$-adhesive). A category $\mathbf{C}$ is $\mathcal{M},\mathcal{N}$-adhesive, where $\mathcal{M}$ is a class of monomorphisms and $\mathcal{N}$ a class of morphisms, if the following properties are satisfied:

1. $\mathcal{M}$ and $\mathcal{N}$ contain all isomorphisms and are closed under composition and decomposition. Moreover, $\mathcal{N}$ is closed under $\mathcal{M}$-decomposition, that is, $f;g \in \mathcal{N}$, $g \in \mathcal{M}$ implies $f \in \mathcal{N}$.
2. $\mathbf{C}$ has $\mathcal{M},\mathcal{N}$-pushouts and $\mathcal{M}$-pullbacks. Also, $\mathcal{M}$ and $\mathcal{N}$ are stable under pushouts and pullbacks.
3. $\mathcal{M},\mathcal{N}$-pushouts are $\mathcal{M},\mathcal{N}$-van Kampen squares.

Remark. $\mathbf{C}$ has *$\mathcal{M},\mathcal{N}$-pushouts*, if there is a pushout whenever one of the given morphisms is in $\mathcal{M}$ and the other morphism is in $\mathcal{N}$. $\mathbf{C}$ has *$\mathcal{M}$-pullbacks*, if there exists a pullback whenever at least one of the given morphisms is in $\mathcal{M}$. A class $\mathcal{X} \in \{\mathcal{M},\mathcal{N}\}$ is *stable under $\mathcal{M},\mathcal{N}$-pushouts* if, given the $\mathcal{M},\mathcal{N}$-pushout (1) in the diagram below, $m \in \mathcal{X}$ implies $n \in \mathcal{X}$ and *stable under $\mathcal{M}$-pullbacks* if, given the $\mathcal{M}$-pullback (1) in the diagram below, $n \in \mathcal{X}$ implies $m \in \mathcal{X}$. An $\mathcal{M},\mathcal{N}$-pushout is an *$\mathcal{M},\mathcal{N}$-van Kampen square* if for the commutative cube (2) in the

diagram below with the pushout (1) as bottom square, $b, c, d, m \in \mathcal{M}$, $f \in \mathcal{N}$, and the back faces being pullbacks, we have that the top square is a pushout if and only if the front faces are pullbacks.



In [HP12], it is shown that all $\mathcal{M}$-adhesive categories are $\mathcal{M},\mathcal{N}$-adhesive.

Lemma 1 ($\mathcal{M}$-adhesive $\Rightarrow$ $\mathcal{M},\mathcal{N}$-adhesive). Let $\mathbf{C}$ be any category and $\mathcal{N}$ be the class of all morphisms in $\mathbf{C}$. Then $\mathbf{C}$ is $\mathcal{M},\mathcal{N}$-adhesive if and only if $\mathbf{C}$ is $\mathcal{M}$-adhesive.

In the following, we give some examples of categories that are $\mathcal{M},\mathcal{N}$-adhesive.

Lemma 2 (Basic $\mathcal{M},\mathcal{N}$-adhesive Categories). The following categories are $\mathcal{M}$-adhesive [EEPT06b] and, by Lemma 1, $\mathcal{M},\mathcal{N}$-adhesive where $\mathcal{N}$ is the class of all morphisms in $\mathbf{C}$:

1. The category **Sets** of sets and functions is $\mathcal{M}$-adhesive where $\mathcal{M}$ is the class of all injective functions.
2. The category **Graphs** of graphs and graph morphisms is $\mathcal{M}$-adhesive where $\mathcal{M}$ is the class of all injective graph morphisms.
3. The category **LGraphs** of labelled graphs and graph morphisms is $\mathcal{M}$-adhesive where $\mathcal{M}$ is the class of all injective graph morphisms.

The following category is $\mathcal{M},\mathcal{N}$-adhesive, but not $\mathcal{M}$-adhesive [HP12]:

4. The category **PLGraphs** of partially labelled graphs and graph morphisms is $\mathcal{M},\mathcal{N}$-adhesive where $\mathcal{M}$ and $\mathcal{N}$ are the classes of all injective and all (injective) undefinedness-preserving[1] graph morphisms, respectively.

3 Construction of Categories

There are various ways to construct new categories from given ones. Beside the standard constructions (product, slice and coslice, functor and comma category) we consider the constructions of a string category and a multiset category. For each of these constructions, we prove a composition result, saying

[1] A morphism $f\colon G \to H$ *preserves undefinedness*, if it maps unlabelled items in G to unlabelled items in H.

more or less, whenever we start with $\mathcal{M}_i, \mathcal{N}_i$-adhesive categories, then the new constructed category is $\mathcal{M}, \mathcal{N}$-adhesive for some $\mathcal{M}, \mathcal{N}$. For the definitions of category-theoretic notions refer to [EEPT06b, Awo10].

First, we consider the standard constructions: product, slice and coslice, functor, and comma category. For the definitions we refer to [EEPT06b] A2 and A6. Our composition result generalizes the result from $\mathcal{M}$- to $\mathcal{M}, \mathcal{N}$-adhesive categories.

Theorem 1 (Standard Constructions). $\mathcal{M}, \mathcal{N}$-adhesive categories can be constructed as follows:

1. If $\mathbf{C_i}$ is $\mathcal{M}_i, \mathcal{N}_i$-adhesive ($i = 1, 2$), then the product category $\mathbf{C_1} \times \mathbf{C_2}$ is $\mathcal{M}, \mathcal{N}$- adhesive where $\mathcal{M} = \mathcal{M}_1 \times \mathcal{M}_2$ and $\mathcal{N} = \mathcal{N}_1 \times \mathcal{N}_2$.
2. If $\mathbf{C}$ is $\mathcal{M}, \mathcal{N}$-adhesive and X is an object of $\mathbf{C}$, then the slice category $\mathbf{C}\backslash X$ and the coslice category $X\backslash\mathbf{C}$ over X are $\mathcal{M}', \mathcal{N}'$-adhesive where the morphism classes $\mathcal{M}', \mathcal{N}'$ are restricted to the slice and coslice category, i.e., for $\mathcal{X} \in \{\mathcal{M}, \mathcal{N}\}$, $\mathcal{X}' = \mathcal{X} \cap \mathbf{C}\backslash X$ and $\mathcal{X}' = \mathcal{X} \cap X\backslash\mathbf{C}$, respectively.
3. If $\mathbf{C}$ is $\mathcal{M}, \mathcal{N}$-adhesive, then for every category $\mathbf{X}$, the functor category $[\mathbf{X}, \mathbf{C}]$ is $\mathcal{M}_{\text{ft}}, \mathcal{N}_{\text{ft}}$-adhesive with functor transformations $\mathcal{M}_{\text{ft}}$ and $\mathcal{N}_{\text{ft}}$.[2]
4. If $\mathbf{C_i}$ are $\mathcal{M}_i, \mathcal{N}_i$-adhesive and $F_i \colon \mathbf{C_i} \to \mathbf{C}$ functors ($i = 1, 2$), where F_1 preserves $\mathcal{M}_1, \mathcal{N}_1$-pushouts and F_2 preserves $\mathcal{M}_2$-pullbacks, then the comma category $\mathbf{ComCat}(F_1, F_2, \mathcal{I})$ is $\mathcal{M}^c, \mathcal{N}^c$-adhesive where $\mathcal{M}^c = (\mathcal{M}_1 \times \mathcal{M}_2) \cap \text{Mor}$, $\mathcal{N}^c = (\mathcal{N}_1 \times \mathcal{N}_2) \cap \text{Mor}$, and Mor is the set of all morphisms of the comma category. We will use $A \downarrow B$ as a shorthand for the comma category $\mathbf{ComCat}(A, B, \mathcal{I})$, with $|\mathcal{I}| = 1$ and both functors A, B pointing into $\mathbf{Sets}$.

Proof. The proof is a slight generalization of the corresponding one for $\mathcal{M}$-adhesive categories (see Theorem 4.15 in [EEPT06b]). In most cases the relevant constructions can be done componentwise from objects or morphisms in the original category. For the full proof see the long version [PH15]. □

Second, we consider the constructions of a string and a multiset category and prove that $\mathcal{M}, \mathcal{N}$-adhesive categories are closed under these constructions.

Construction (String Category). Given a category $\mathbf{C}$, we construct a *string category* $\mathbf{C}^*$ as follows:

The objects are lists (finite sequences) $A_1 \dots A_m$ of objects of $\mathbf{C}$, including the empty list λ. The morphisms between two objects $A_1 \dots A_m$ and $B_1 \dots B_n$ (given $m \leq n$) are lists (finite sequences) of morphisms $f_1 \colon A_1 \to B_i \dots f_m \colon A_m \to B_{i+m-1}$ in $\mathbf{C}$, with $B_1 \dots B_i \dots B_{i+m-1} \dots B_n$ for some $1 \leq i \leq m-n$ (i.e. $A_1 \dots A_m$ is *embedded* in $B_1 \dots B_n$). The empty list λ is an initial element for $\mathbf{C}^*$.

Our construction for a string category above is close to that of a free monoidal category. Allowing for the existence of a morphism even if $m < n$, however

[2] For a class $\mathcal{X}$, $\mathcal{X}_{\text{ft}}$ denotes the class of natural transformations $t \colon F \to G$, where all morphisms $t_X \colon F(X) \to G(X)$ are in $\mathcal{X}$.

contradicts these definitions and further prevents us from giving a workable definition of a tensor product. We need these morphisms, especially in the case of the multiset category below, to allow for the addition or removal of elements in transformation systems based on these categories.

Construction (Multiset Category). Given a category $\mathbf{C}$, we construct a *multiset category* $\mathbf{C}^{\oplus}$ as follows:

The object are lists (finite sequences) $A_1 \dots A_m$ of objects of $\mathbf{C}$, including an empty list $\emptyset$. The morphisms between two objects $A_1 \dots A_m$ and $B_1 \dots B_n$ (given $m \leq n$) are lists (finite sequences) of morphisms $f_i \colon A_i \to B_{j_i}$ in $\mathbf{C}$, where $j_i = j_k$ implies $i = k$, $i \in \{1, \dots, m\}$.In contrast to the above construction for a string category, we ignore the order of elements.

We will use $\{\!|a, a, b|\!\}$ to denote a multiset with elements a, a and b.

Theorem 2 ($\mathbf{C}$ $\mathcal{M},\mathcal{N}$-adh $\Rightarrow$ $\mathbf{C}^*$ $\mathcal{M}^*,\mathcal{N}^*$-adh, $\mathbf{C}^{\oplus}$ $\mathcal{M}^{\oplus},\mathcal{N}^{\oplus}$-adh).

1. If $\mathbf{C}$ is $\mathcal{M},\mathcal{N}$-adhesive, then the string category $\mathbf{C}^*$ over $\mathbf{C}$ is $\mathcal{M}^*,\mathcal{N}^*$-adhesive where $\mathcal{M}^*$ and $\mathcal{N}^*$ contain those morphisms which are lists of morphisms in $\mathcal{M}$ and $\mathcal{N}$, respectively. $\mathcal{N}^*$ is further restricted to morphisms that perserve length, i.e. where domain and codomain are of equal length.
2. If $\mathbf{C}$ is $\mathcal{M},\mathcal{N}$-adhesive, then the multiset category $\mathbf{C}^{\oplus}$ over $\mathbf{C}$ is $\mathcal{M}^{\oplus},\mathcal{N}^{\oplus}$-adhesive with $\mathcal{M}^{\oplus}$ and $\mathcal{N}^{\oplus}$ contain those morphisms which are lists of morphisms in $\mathcal{M}$ and $\mathcal{N}$, respectively.

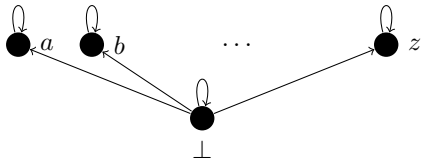
Proof. In both cases the relevant constructions can be done componentwise from objects or morphisms in the original category and the composition and decomposition properties can be inherited from morphisms in the underlying category. The restriction of $\mathcal{N}^*$ to morphisms that preserve length ensures the existence of pushouts. See the long version [PH15] for the full proof. □

4 Partially Labelled Graphs

Let us reconsider the category **PLGraphs** of partially labelled graphs, investigated e.g. in [HP02, HP12], where the labelling functions for nodes and edges are allowed to be partial. In [HP12], it is shown that **PLGraphs** is not $\mathcal{M}$-adhesive, but $\mathcal{M},\mathcal{N}$-adhesive if we choose $\mathcal{M}$ and $\mathcal{N}$ as the classes of all injective and all injective, undefinedness-preserving graph morphisms, respectively. In this section, we present an alternative proof of the statement: We show that the category **PLGraphs** can be constructed from the category **Graphs** and a category **PL** of labels by a multiset construction and the comma category construction.

First, we consider a label set L together with the symbol $\perp$ indicating undefinedness. As morphisms we use all identities as well as all morphisms from $\perp$ to a label in L.

Lemma 3 (PL is a Category). For each alphabet L, the class of all elements in $L \cup \{\bot\}$[3] as objects and all morphisms of the form $\bot \to x$ and $x \to x$ ($x \in L \cup \{\bot\}$) forms the category **PL** where the composition of $x \to y$ and $y \to z$ is $x \to z$ and the identity on x is $x \to x$.



Proof. Follows directly from the definiton. □

It can be shown that the category **PL** is $\mathcal{M},\mathcal{N}$-adhesive.

Lemma 4 (PL is $\mathcal{M},\mathcal{N}$-adhesive). The category **PL** is $\mathcal{M},\mathcal{N}$-adhesive where $\mathcal{M}$ and $\mathcal{N}$ are the classes of all morphisms and all identities, respectively.

Proof. We show the properties required in Definition 1, for pushouts and van Kampen squares we can list the small number of cases exhaustively. See the long version [PH15] for the full proof. □

Partially labelled graphs generalize labelled graphs [Ehr79].

Definition 2 (PLGraphs). A *partially labelled graph* is a system $G = (V, E, s, t, l)$ consisting of finite sets V and E of nodes and edges, source and target functions $s, t\colon E \to V$, and a partial labelling function $l\colon E + V \to L$ [4], where L is a fixed set of labels.

A *morphism* $g\colon G \to H$ between graphs G and H consists of two functions $g_V\colon V_G \to V_H$ and $g_E\colon E_G \to E_H$ that preserve sources, targets and labels, that is, $g_E; s_H = s_G; g_V$, $g_E; t_H = t_G; g_V$, and $l_H(g(x)) = l_G(x)$ for all x in $\mathrm{Dom}(l_G)$.

Fact 1. The class of partially labelled graphs and its morphisms constitute a category **PLGraphs**, where morphism composition is function composition and the identity is the identity function.

As an alternative to the existing proof we prove that the comma category of the two functors $Graphs\colon$ **Graphs** $\to$ **Sets** and $PL\colon \mathbf{PL}^{\oplus} \to$ **Sets** defined below is $\mathcal{M},\mathcal{N}$-adhesive. We further prove the category **PLGraphs** is isomorphic to this comma category, thus **PLGraphs** is also $\mathcal{M},\mathcal{N}$-adhesive. The isomorphism of categories is defined as in Ehrig et. al. [EEPT06b].

[3] We assume that $\bot$ is not an element of L.
[4] $+$ denotes the disjoint union of sets.

Definition 3 (*Graphs*: Graphs → Sets). The functor *Graphs*: **Graphs** → **Sets** maps graphs to their underlying set of nodes and edges and is given as follows: For a graph $G' = (V', E', s', t')$, let $Graphs(G') = V' + E'$ and for a graph morphism $f_{G'}: A \to B$, let $Graphs(f_{G'})$ be a natural transformation, defined by $Graphs(f)(x) = f_{V'}(x)$ if $x \in V'$ and $f_{E'}(x)$ otherwise.

Lemma 5. The functor *Graphs*: **Graphs** → **Sets** preserves $\mathcal{M},\mathcal{N}$-pushouts, where $\mathcal{M}$ is the class of injective graph morphisms, $\mathcal{N}$ is the class of all morphisms.

Proof. See the long version [PH15]. □

Definition 4 (PL: $\mathbf{PL}^{\oplus}$ → Sets). The functor PL: $\mathbf{PL}^{\oplus}$ → **Sets** maps a multiset of labels to a set with distinct elements and is given as follows: For a multiset of labels $m' : L' \to \mathbb{N}$ let $PL(m) = \bigcup_{l' \in L'} \bar{m}(l')$, where for $l' \in L'$, $\bar{m}(l') = \{l'_1, ..., l'_k\}$ iff $m(l') = k$. For a morphism $f: m_1 \to m_2$ let $PL(f) = PL(m_1) \to PL(m_2)$ be a morphism in **Sets**, such that $PL(f)(l'_1) = l'_2$ iff $PL(l_1) = l'_1, PL(l_2) = l'_2$ and $f(l_1) = l_2$ with $l_1, l_2 \in m_1, m_2$ respectively.

Lemma 6. The functor PL: $\mathbf{PL}^{\oplus}$ → **Sets** preserves $\mathcal{I}$-pullbacks, where $\mathcal{I}$ is the class of all morphisms.

Proof. See the long version [PH15]. □

For an object (G', m', op) of the comma category $Graphs \downarrow PL$ the morphism $\mathrm{op}: Graphs(G') \to PL(m')$ determines which node or edge is associated with which label, i.e. $\mathrm{op}(e') = l'_i$ with $e' \in E', l' \in L'$ and $i \in \mathbb{N}$ means the edge e' is labelled with l'.

Lemma 7 (PLGraphs ≅ $Graphs \downarrow_s PL$). The category **PLGraphs** of partially labelled graphs is isomorphic to the comma category $Graphs \downarrow_s PL$, where $\downarrow_s$ indicates a restriction to surjective morphisms op in the comma category.

We restrict ourselves to those objects of the comma category where op is surjective, since there could otherwise be labels that are not associated with an object in the graph.

Proof. The graph components of both categories are trivially isomorphic. It remains to show that changes to a partial labelling function and a total labelling function along with changes to the labels themselves can be employed towards the same effect. For the proof see the long version [PH15]. □

Now we are able to present an alternative proof of the fact that the category **PLGraphs** of partially labelled graphs is $\mathcal{M},\mathcal{N}$-adhesive. It is based on fact that the categories **Graphs** of graphs and **PL** of labels are $\mathcal{M},\mathcal{N}$-adhesive and the constructions of a commutative monoidal category and the comma category preserve $\mathcal{M},\mathcal{N}$-adhesiveness.

Theorem 3 (PLGraphs is $\mathcal{M},\mathcal{N}$-adhesive). The category **PLGraphs** of partially labelled graphs is $\mathcal{M},\mathcal{N}$-adhesive where $\mathcal{M}$ and $\mathcal{N}$ are the classes of all injective and all injective, undefinedness-preserving graph morphisms, respectively.

Proof. The new proof of Theorem 3 is illustrated in Figure 1.

1. By Lemmata 2 and 4, **Graphs** and **PL** are $\mathcal{M}_G,\mathcal{N}_G$ and $\mathcal{M}_L,\mathcal{N}_L$-adhesive, respectively. $\mathcal{M}_G,\mathcal{N}_G$ are monomorphisms in **Graphs** and $\mathcal{M}_L,\mathcal{N}_L$ are the classes of all morphisms and all identity morphisms in **PL**, respectively.
2. By Theorem 2, $\mathbf{PL}^{\oplus}$ is $\mathcal{M}^{\oplus},\mathcal{N}^{\oplus}$-adhesive.
3. By Theorem 1 and Lemmata 5 and 6, $Graphs \downarrow_s PL$ is $\mathcal{M}^c,\mathcal{N}^c$-adhesive. Note that Theorem 1 still holds for the restriction to a surjective op, since the componentwise constructions can be still be done just as in the unrestricted case.
4. By Lemma 7, **PLGraphs** is $\mathcal{M},\mathcal{N}$-adhesive. Moreover $\mathcal{M} = F(\mathcal{M}^c)$ since both of these classes include all monomorphisms and $\mathcal{N} = F(\mathcal{N}^c)$ since the perservation of undefinedness in $\mathcal{N}$ is analogous to the restriction to identity morphisms in $\mathcal{N}^L$, which determines the treatment of labels in $\mathcal{N}^c$.

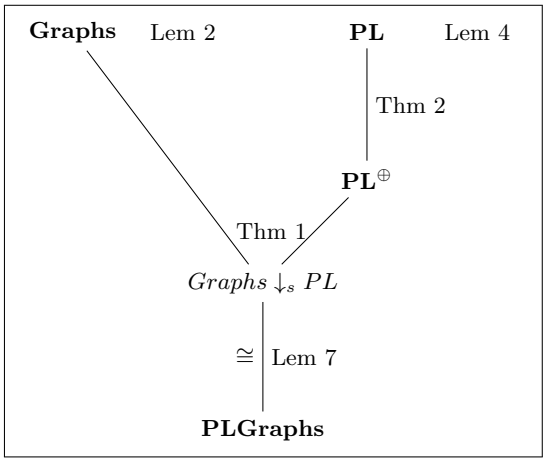


Fig. 1. Proof of "**PLGraphs** is $\mathcal{M},\mathcal{N}$-adhesive".

□

Example 1. Figure 2 shows a transformation rule for a partially labelled graph consisting of a single node which is relabelled from a to b. Below the rule we show objects of $Graphs \downarrow PL$ and their individual components. Note that, in contrast to partially labelled graphs, we do not change the assigment of items to labels but instead change the label itself.

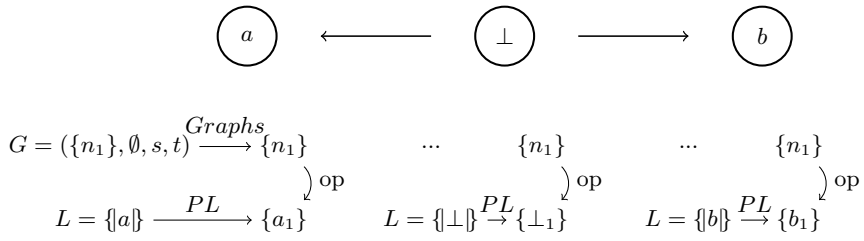


Fig. 2. Example transformation of an object of $Graphs \downarrow_s PL$

5 Attributed Graphs

Similiar to the construction for partially labelled graphs we construct attributed graphs, where the attributes can be changed analogously to relabelling.

We start with defining a category where the objects collect all the attributes of a node or an edge. These *attribute collections* consist of a set of names, each of which is associated with a value. We use the category **PL** from section 4 to represent these values.

Definition 5 (Att $= ID_{\mathbf{Sets}} \downarrow PL$). The category **Att** of attribute collections is defined as the comma category $ID_{\mathbf{Sets}} \downarrow PL$ where $ID_{\mathbf{Sets}}$ denotes the identity functor over **Sets**.

Lemma 8 (Att is $\mathcal{M}^c, \mathcal{N}^c$-adhesive). The category **Att** of attribute collections is $\mathcal{M}^c, \mathcal{N}^c$-adhesive where $\mathcal{M}^c, \mathcal{N}^c$ are the classes of morphisms induced by the comma category construction.

Proof. The proof is illustrated in Figure 3. $ID_{\mathbf{Sets}} \downarrow PL$ is $\mathcal{M}^c, \mathcal{N}^c$-adhesive, since **Sets** and **PL** are $\mathcal{M}, \mathcal{N}$-adhesive and a multiset and comma category construction preserve $\mathcal{M}, \mathcal{N}$-adhesiveness. □

To construct attributed graphs we define a functor from multisets of these attribute collections to sets for later use in the comma category construction (compare with the construction for partially labelled graphs in section 4). We also prove that this functor preserves pullbacks, since this is required for the comma category to preserve $\mathcal{M}, \mathcal{N}$-adhesiveness.

Definition 6 ($Att\colon \mathbf{Att}^{\oplus} \to \mathbf{Sets}$). The functor $Att\colon \mathbf{Att}^{\oplus} \to \mathbf{Sets}$ that maps attribute collections to sets with distinct values is given by the following: A triple $(ID_{\mathbf{Sets}}(S), PL(m), \mathrm{op})^{\oplus}$ is mapped to the set $S^{\oplus} + PL(m)^{\oplus}$ where $\oplus$ is flattened analogously to the way PL does (see Definition 4). A morphism $f\colon A \to B$ in $\mathbf{Att}^{\oplus}$ is mapped to a morphism $Att(f)\colon Att(A) \to Att(B)$, where elements in $Att(A)$ are mapped to elements in $Att(B)$ based on the original mappings in $\mathbf{Att}^{\oplus}$.

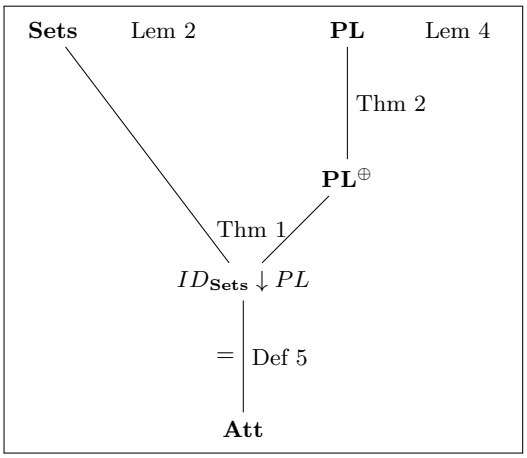


Fig. 3. Proof of "**Att** is $\mathcal{M}^c, \mathcal{N}^c$-adhesive".

Lemma 9 (*Att* preserves $\mathcal{I}$-pullbacks). The functor $Att\colon \mathbf{Att}^{\oplus} \to \mathbf{Sets}$ preserves $\mathcal{I}$-pullbacks where $\mathcal{I}$ is the class of all morphisms.

Proof. See the long version [PH15]. □

Example 2. Figure 4 shows a transformation rule in **Att**. The attribute collection consists of a single attribute a, which has its value changed from 5 to 9 by the rule. Below the rule we show the objects of **Att** with their individual components.

$$a = 5 \longleftarrow a = \bot \longrightarrow a = 9$$

$$\begin{array}{ccccccc} N = \{a\} & \xrightarrow{ID_{\mathbf{Sets}}} & \{a\} & \cdots & \{a\} & \cdots & \{a\} \\ \downarrow \text{op} & & \downarrow \text{op} & & \downarrow \text{op} & & \downarrow \text{op} \\ V = \{\!|5|\!\} & \xrightarrow{PL} & \{5_1\} & V = \{\!|\bot|\!\} & \xrightarrow{PL} & \{\bot_1\} & V = \{\!|9|\!\} \xrightarrow{PL} \{9_1\} \end{array}$$

Fig. 4. Example transformation rule for objects of **Att**

Definition 7 (AttGraphs = *Graphs* ↓ *Att*). The category **AttGraphs** of attributed graphs is defined as the comma category *Graphs* ↓ *Att*.

Now we are able to show that the category **AttGraphs** of attributed graphs is $\mathcal{M}, \mathcal{N}$-adhesive. It is based on the fact that the categories **Graphs** of graphs

and **Att** of attribute collections are $\mathcal{M},\mathcal{N}$-adhesive and the constructions of a multiset category and a comma category preserve $\mathcal{M},\mathcal{N}$-adhesiveness.

Theorem 4 (AttGraphs is $\mathcal{M},\mathcal{N}$-adhesive). The category **AttGraphs** of attributed graphs is $\mathcal{M}^c,\mathcal{N}^c$-adhesive where $\mathcal{M}^c,\mathcal{N}^c$ are the classes of morphisms induced by the comma category construction.

Proof. The proof is illustrated in Figure 5.

1. By Lemmata 2 and 8, **Graphs** and **Att** are $\mathcal{M}_G,\mathcal{N}_G$ and $\mathcal{M}_A,\mathcal{N}_A$-adhesive, respectively. $\mathcal{M}_G,\mathcal{N}_G$ are monomorphisms in **Graphs** and $\mathcal{M}_A,\mathcal{N}_A$ are the classes of morphisms induced by the comma category construction of **Att**.
2. By Theorem 2, $\mathbf{Att}^{\oplus}$ is $\mathcal{M}^{\oplus},\mathcal{N}^{\oplus}$-adhesive.
3. By Theorem 1 and Lemmata 5 and 9, $Graphs \downarrow Att$ is $\mathcal{M}^c,\mathcal{N}^c$-adhesive.
4. By Defintion 7, **AttGraphs** is $\mathcal{M}^c,\mathcal{N}^c$-adhesive.

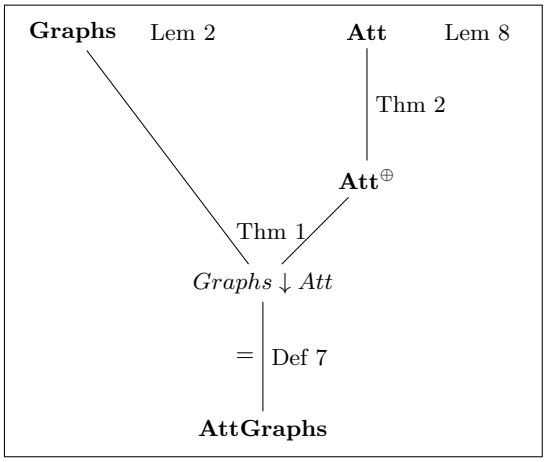


Fig. 5. Proof of "**AttGraphs** is $\mathcal{M}^c,\mathcal{N}^c$-adhesive".

□

In the following we briefly compare these attributed graphs to some existing attribution concepts. In contrast to the typed attributed graphs in [EEPT06b] these attributed graphs can have at most one value for an attribute. We constructed untyped graphs and even the attributes themselves have no types. Using the construction for the slice category from Theorem 1 we can build graphs where typing is done at either level, which allows us to have typed attributes on an untyped graphs, such that attribute values are constrained by the type but what

attributes a node or edge has is not. In contrast typed attributed graphs require that the graph is typed and thus do not allow e.g. the addition of attributes to a node or edge. Compared to the attribution concepts used in [Plu09] we do not require a seperate instantiation of a rule schema and it is possible to find a match without fully specifying other, pontentially uninteresting, attributes. We do not, however base our attributes on an algebra that would allow us to perform some computations on the attributes, this would require additional work to prove $\mathcal{M},\mathcal{N}$-adhesiveness for a suitable category. Fortunately we only need to provide this proof for such attributes once, enabling us to construct many different attributed structures without concerning ourselves with e.g. the underlying graphs.

6 $\mathcal{W}$-adhesive Categories

The concept of $\mathcal{M},\mathcal{N}$-adhesive categories [HP12] was introduced as a framework for partially labeled graphs. More or less at the same time, the concept of $\mathcal{W}$-adhesive categories was introduced by Golas [Gol12] as a framework for attributed graphs. In this section, we present a precise relationship between $\mathcal{M},\mathcal{N}$-adhesive and $\mathcal{W}$-adhesive categories.

We obtain the following relationship between $\mathcal{M},\mathcal{N}$- and $\mathcal{W}$-adhesive categories.

Theorem 5 ($\mathcal{M},\mathcal{N}$-adhesive $\Rightarrow$ $\mathcal{W}$-adhesive). If the category $\mathbf{C}$ is $\mathcal{M},\mathcal{N}$-adhesive, then the tuple $(\mathbf{C},\mathcal{M},\mathcal{M},\mathcal{M}\times\mathcal{N})$ is a $\mathcal{W}$-adhesive category. Vice versa, if the tuple $(\mathbf{C},\mathcal{R},\mathcal{M},\mathcal{W})$ is $\mathcal{W}$-adhesive, then $\mathbf{C}$ is $\mathcal{R},\mathrm{Ran}(\mathcal{W})$-adhesive provided that the range $\mathrm{Ran}(\mathcal{W})$ of $\mathcal{W}$ is stable under pushout and pullback and contains all isomorphisms.

Proof. We can directly derive the properties of $\mathcal{W}$-adhesive categories from the definition of $\mathcal{M},\mathcal{N}$-adhesive categories and vice versa, except for the stability of the morphism classes $\mathcal{M},\mathcal{N}$ over pushouts and pullbacks. Due to the required properties of $\mathcal{W}$-adhesive categories, the class of $\mathcal{W}$-spans used necessarily equals the spans defined by $\mathcal{R}\times\mathcal{N}$ thus allowing us to bridge the different definitions. For the proof see the long version [PH15]. □

Remark. The situation may be summarized as follows:

- The requirements for an $\mathcal{M},\mathcal{N}$-adhesive category are slightly more strict than those for $\mathcal{W}$-adhesive categories.
- For $\mathcal{M},\mathcal{N}$-adhesive systems, the Local Church-Rosser Theorem, the Parallelism Theorem, and the Concurrency Theorem are proven. For $\mathcal{W}$-adhesive systems, up to our knowledge, there has only been a proof of (part of) the Local Church-Rosser Theorem.
- The $\mathcal{W}$-adhesive categories of attributed objects in [Gol12] are $\mathcal{M},\mathcal{N}$-adhesive: $\mathcal{N}$ is the class of $\perp$-preserving morphisms, contains all isomorphisms and is stable under pushout and pullbacks.

7 Related Concepts

Throughout the literature, various versions of adhesive and quasiadhesive [LS05], weak adhesive HLR [EHPP06], partial map adhesive [Hei10], and $\mathcal{M}$-adhesive [EGH10] exist. In [EGH10], all these categories are shown to be also $\mathcal{M}$-adhesive ones. The categories of labelled graphs, typed graphs, and typed attributed graphs in [EEPT06b], are known to be $\mathcal{M}$-adhesive categories if one chooses $\mathcal{M}$ to be the class of injective graph morphisms [EGH10]. Each such category induces a class of $\mathcal{M}$-adhesive systems for which several classical results of the double-pushout approach hold.

Unfortunately, the framework of $\mathcal{M}$-adhesive systems does not cover graph transformation with relabelling. In [HP12], the authors generalize $\mathcal{M}$-adhesive categories to $\mathcal{M},\mathcal{N}$-adhesive categories, where $\mathcal{N}$ is a class of morphisms containing the vertical morphisms in double-pushouts, and show that the category of partially labelled graphs is $\mathcal{M},\mathcal{N}$-adhesive, where $\mathcal{M}$ and $\mathcal{N}$ are the classes of injective and injective, undefinedness-preserving graph morphisms, respectively. Independently, Golas [Gol12] provided a general framework for attributed objects, so-called $\mathcal{W}$-adhesive systems which allows undefined attributes in the interface of a rule to change attributes, which is similar to relabelling. By Lemma 1 and Theorem 5, the hierarchy of adhesive categories in [EGH10] can be extended in the following way:

$$\text{adhesive} \overset{\Rightarrow}{\not\Leftarrow} \text{adhesive HLR} \overset{\Rightarrow}{\not\Leftarrow} \mathcal{M}\text{-adhesive} \overset{\Rightarrow}{\not\Leftarrow} \mathcal{M},\mathcal{N}\text{-adhesive} \overset{\Rightarrow}{\not\Leftarrow} \mathcal{W}\text{-adhesive}$$

In the literature, there are several variants of attribution concepts, e.g., Löwe et al. [LKW93] view graphs as a special case of algebras. These algebras can then additionally specify types for attributes. Ehrig et al. [EEPT06a] — introduce typed attributed graphs, expanding the graph by including an algebra for attribute values. To facilitate attribution, typed attributed graphs extend graphs by attribution nodes and attribution edges. All possible data values of the algebra are assumed to be part of the graph. Nodes and edges are attributed by adding an attribution edge that leads to an attribution node. Kastenberg and Rensink [KR12] take a similar approach, but instead of only encoding the data values, operations and constants are also included in the graph. Plump et al [Plu09] use a different approach to attribution. Here labels are replaced by sequences of attributes. Rules are complemented by rule schemata in which terms over the attributes are specified. These variables are substituted with attribute values and evaluated during rule application. Instead of modifying the definition of graphs and graph transformations to include attributes, Golas [Gol12] defines an attribution concept over arbitrary categories. Duval et al. [DEPR14] allow attributed graphs and allow rules to change attributes.

In [Peu13], Peuser compares the approaches of Ehrig et al. [EEPT06a] and Plump [Plu09] and introduces a useful new concept of attribution which is the basis of this work.

The idea of composition of adhesive categories is not new: For $\mathcal{M}$-adhesive categories, the standard constructions of product, slice and coslice, functor, and comma categories are given in [EEPT06b].

8 Conclusion

In this paper, we have continued the work on $\mathcal{M},\mathcal{N}$-adhesive categories and have presented several examples (see Table 1).

category	structures	adhesiveness	reference
Sets	sets	$\mathcal{M}$-adh	[EEPT06b]
PL	sets of labels	$\mathcal{M},\mathcal{N}$-adh	Lemma 4
Att	attribute collections	$\mathcal{M},\mathcal{N}$-adh	Lemma 8
Graphs	unlabelled graphs	$\mathcal{M}$-adh	[EEPT06b]
LGraphs	labelled graphs	$\mathcal{M}$-adh	[Ehr79]
PLGraphs	partially labelled graphs	$\mathcal{M},\mathcal{N}$-adh	[HP12], Thm 3
AttGraphs	attributed graphs	$\mathcal{M},\mathcal{N}$-adh	Theorem 4

Table 1. Examples of $\mathcal{M},\mathcal{N}$-adhesive categories

The main contributions of the paper are the following:

(1) Closure results for $\mathcal{M},\mathcal{N}$-adhesive categories.
(2) A new, shorter proof of the result in [HP12] that the category of partially labeled graphs is $\mathcal{M},\mathcal{N}$-adhesive.
(3) A new concept of attributed graphs together with a proof that the category of these attributed graphs is $\mathcal{M},\mathcal{N}$-adhesive and an application to transformation systems saying that for these attributed graphs, the Local Church-Rosser Theorem, the Parallelism Theorem and the Concurrency Theorem hold provided that the HLR$^+$-properties are satisfied.

Further topics might be:

(1) Investigate the relationship to the approach of Parisi-Presicce et al. [PEM87] considering graphs with a structured alphabet.
(2) Proof of the HLR$^+$-properties for the category **AttGraphs** to obtain the Local Church-Rosser Theorem, the Parallelism Theorem and the Concurrency Theorem for this type of attributed graphs.
(3) Generalization of the approach to systems with so-called left-linear rules, i.e., rules where only the left morphism of the rule is required to be in $\mathcal{M}$ as, e.g., in [BGS11].

Acknowledgements. We would like to thank the anonymous referees and Wolfram Kahl for helpful feedback on an early version of this paper.

References

Awo10. Steve Awodey. *Category Theory*. Oxford University Press, 2010.

BGS11. Paolo Baldan, Fabio Gadducci, and Paweł Sobociński. Adhesivity is not enough: Local Church-Rosser revisited. In *Mathematical Foundations of Computer Science (MFCS 2011)*, volume 6907 of *Lecture Notes in Computer Science*, pages 48–59, 2011.

DEPR14. Dominique Duval, Rachid Echahed, Frederic Prost, and Leila Ribeiro. Transformation of attributed structures with cloning. In *Fundamental Approaches to Software Engineering*, volume 8411 of *Lecture Notes in Computer Science*, pages 310–324, 2014.

EEKR99. Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2: Applications, Languages and Tools. World Scientific, 1999.

EEPT06a. Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental theory of typed attributed graph transformation based on adhesive HLR categories. *Fundamenta Informaticae*, 74(1):31–61, 2006.

EEPT06b. Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs of Theoretical Computer Science. Springer, 2006.

EGH10. Hartmut Ehrig, Ulrike Golas, and Frank Hermann. Categorical Frameworks for Graph Transformation and HLR Systems based on the DPO Approach. *Bulletin of the EATCS*, 112:111–121, 2010.

EHKP91. Hartmut Ehrig, Annegret Habel, Hans-Jörg Kreowski, and Francesco Parisi- Presicce. From graph grammars to high level replacement systems. In *Graph Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 269–291, 1991.

EHPP06. Hartmut Ehrig, Annegret Habel, Julia Padberg, and Ulrike Prange. Adhesive high-level replacement systems: A new categorical framework for graph transformation. *Fundamenta Informaticae*, 74(1):1–29, 2006.

Ehr79. Hartmut Ehrig. Introduction to the Algebraic Theory of Graph Grammars. In *Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 1–69, 1979.

EKMR99. Hartmut Ehrig, Hans-Jörg Kreowski, Ugo Montanari, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 3: Concurrency, Parallelism, and Distribution. World Scientific, 1999.

Gol12. Ulrike Golas. A General Attribution Concept for Models in $\mathcal{M}$-adhesive Transformation Systems. In *Graph Transformations (ICGT'12)*, volume 7562 of *Lecture Notes in Computer Science*, pages 187–202, 2012.

Hei10. Tobias Heindel. Hereditary pushouts reconsidered. In *Graph Transformations (ICGT 2010)*, volume 6372 of *Lecture Notes in Computer Science*, pages 250–265, 2010.

HP02. Annegret Habel and Detlef Plump. Relabelling in Graph Transformation. In *Graph Transformation (ICGT 2002)*, volume 2505 of *Lecture Notes in Computer Science*, pages 135–147, 2002.

HP12. Annegret Habel and Detlef Plump. $\mathcal{M},\mathcal{N}$-adhesive Transformation Systems. In *Graph Transformations (ICGT 2012)*, volume 7562 of *Lecture*

Notes in Computer Science, pages 218–233, 2012. Long version available at: `http://formale-sprachen.informatik.uni-oldenburg.de/pub/index.html`.

KR12. Harmen Kastenberg and Arend Rensink. Graph attribution through subgraphs. Technical Report TR-CTIT-12-27, University of Twente, 2012.

LKW93. Michael Löwe, Martin Korff, and Annika Wagner. An algebraic framework for the transformation of attributed graphs. In *Term Graph Rewriting: Theory and Practice*, pages 185–199. John Wiley, 1993.

LS05. Stephen Lack and Paweł Sobociński. Adhesive and quasiadhesive categories. *Theoretical Informatics and Application*, 39(2):511–546, 2005.

PEM87. Francesco Parisi-Presicce, Hartmut Ehrig, and Ugo Montanari. Graph rewriting with unification and composition. In *Graph Grammars and Their Application to Computer Science*, volume 291 of *Lecture Notes in Computer Science*, pages 496–514, 1987.

Peu13. Christoph Peuser. Attribution Concepts for Graph Transformation. Master's thesis, 2013. Available at: `http://formale-sprachen.informatik.uni-oldenburg.de/pub/index.html`.

PH15. Christoph Peuser and Annegret Habel. Attribution of graphs by composition of $\mathcal{M},\mathcal{N}$-adhesive categories: Long version. Available at: `http://formale-sprachen.informatik.uni-oldenburg.de/~skript/fs-pub/PH15_mn-composition_long.pdf`, 2015.

Plu09. Detlef Plump. The graph programming language GP. In *Proc. Algebraic Informatics (CAI 2009)*, volume 5725 of *Lecture Notes in Computer Science*, pages 299–122, 2009.

Roz97. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1: Foundations. World Scientific, 1997.

Single-Pushout Rewriting of Partial Algebras

Michael Löwe * and Marius Tempelmeier †

* Fachhochschule für die Wirtschaft (FHDW) Hannover
† Sennheiser electronic GmbH & Co. KG

Abstract We introduce Single-Pushout Rewriting for *arbitrary* partial algebras. Thus, we give up the usual restriction to graph structures, which are algebraic categories with unary operators only. By this generalisation, we obtain an integrated and straightforward treatment of graphical structures (objects) and attributes (data). We lose co-completeness of the underlying category. Therefore, a rule is no longer applicable at any match. We characterise the new application condition and make constructive use of it in some practical examples.

1 Introduction

The current frameworks for the (algebraic) transformation of typed graphs are not completely satisfactory from the software engineering perspective. For example, it is hardly possible to specify and handle associations with "at-most-one"-multiplicity, since most frameworks are based on some (adhesive) categories of graphs which allow multiple edges between the same pair of vertices.[1]

Another example is the handling of attributes. The standard approaches to the transformation of attributed graphs, compare for example [5,13], explicitly distinguish two parts, i. e. the *structure part* (objects and links) which can be changed by transformations and the base-type and -operation part (data) which is immutable. Typically, objects can be attributed with data via some special edges the source of which is in the structure part and the target of which is data. This set-up either leads to set-valued or immutable attribute structures. Both is not satisfactory from the software engineering point of view.[2]

Another problem in current frameworks for attributed graphs is the infiniteness of rules stipulated by the infiniteness of the term algebra which is typically used in the rules. Even if the algebra for the objects which are subject to transformation is finite (for example integers modulo some maximum), the term algebra tends to contain infinitely many terms.

All these problems are more or less caused by the usage of total algebras. In this paper, we use partial algebras instead as the underlying category for single-pushout rewriting. In partial algebras, operation definitions can be

[1] Typically, some negative application conditions [8] are employed to handle these requirements making the framework more complicated.

[2] In object-oriented programming languages, for example, attributes have the standard multiplicity `0..1`.

changed without deleting and adding an object (edge). Thus, we get a straightforward model for "at-most-one" associations. We also give up the distinction between structure and data, i. e. we allow arbitrary signatures which are able to integrate both parts. We lose co-completeness of the base category and import some application conditions into single-pushout rewriting. But we gain a seamless integration of structure and data. Finally, partial term algebras in the rules help to keep rules finite.

The paper is a short version of [15], where many more examples and details can be found. Section 2 introduces our concept of partial algebra. We show explicitly the similarities between partial algebras and hypergraphs. Section 3 provides sufficient and necessary conditions for the existence of pushouts in categories of partial algebras and partial morphisms. It contains our main results. Section 4 defines the new single-pushout approach and shows similarities and differences to the sesqui-pushout approach [3]. Section 5 demonstrates by some example that the new application conditions are useful in many situations. Finally, Section 6 discusses related work and provides some conclusions.

2 Graphs and Partial Algebras

In this section, we introduce the basic notions and constructions for partial algebras. We use a rather unusual approach in order to emphasise the close connection of categories of partial algebras to categories of hypergraphs. We employ a kind of objectification for partial mappings. A partial map $f : A \to B$ is not just a subset of $A \times B$ satisfying the uniqueness condition $(*)$ $(a, b_1), (a, b_2) \in f$ implies $b_1 = b_2$. Instead, a partial map $f : A \to B$ is a triple $(f, d_f : f \to A, c_f : f \to B)$ consisting of a set f of *map entries* together with two *total* maps $d_f : f \to A$ and $c_f : f \to B$ which provide the argument and the return value for every entry respectively. The uniqueness condition $(*)$ above translates to $\forall e_1, e_2 \in f : d_f(e_1) = d_f(e_2) \Longrightarrow e_1 = e_2$ in this set-up.

A *signature* $\Sigma = (S, O)$ consists of a set of sort names S and a domain- and co-domain-indexed family of operation names $O = (O_{w,v})_{w,v \in S^*}$.[3] A *graph* G wrt. a signature consists of a carrier set G_s (of vertices) for every sort name $s \in S$ and a set of hyperedges $\left(f^G, d_f^G : f^G \to G^w, c_f^G : f^G \to G^v\right)$ for every operation name $f \in O_{w,v}$ and $w, v \in S^*$ where the *total* mappings d_f^G and c_f^G provide the "arguments" and "return values".[4] A *graph morphism* $h : G \to H$ between to graphs G and H wrt. the same signature $\Sigma = (S, O)$ consists of a family of vertex mappings $h = (h_s : G_s \to H_s)_{s \in S}$ and a family of edge mappings $h^O = \left(h_f^O : f^G \to f^H\right)_{f \in O}$ such that for all operation names $f \in O_{w,v}$ and for

[3] Note that we generalise the usual notion of signature which allows single sorts as co-domain specification for operation names only. Operation names in $O_{w,\epsilon}$ will be interpreted as predicates, operation names in $O_{w,v}$ with $|v| > 1$ will be interpreted as operations which deliver several results simultaneously.

[4] For $w \in S^*$ and a family $(G_s)_{s \in S}$ of sets, G^w is recursively defined by (i) $G^\epsilon = \{*\}$, (ii) $G^w = G_s$ if $w = s \in S$ and (iii) $G^w = G^v \times G^u$ if $w = vu$.

all edges $e \in f^G$ the following homomorphism condition holds:[5]

$$\text{(h)}\ \ d_f^H\left(h_f^O(e)\right) = h^w\left(d_f^G(e)\right) \text{ and } c_f^H\left(h_f^O(e)\right) = h^v\left(c_f^G(e)\right).$$

The category of all graphs and graph morphisms wrt. a signature Σ is denoted by $\mathcal{G}_\Sigma$ in the following.[6] $\mathcal{G}_\Sigma$ is complete and co-complete. All limits and co-limits can be constructed component-wise on the underlying sets. The pullback for a co-span $B \overset{m}{\to} A \overset{n}{\leftarrow} C$ is given by the *partial product* $B \times_{(m,n)} C$ with the two *projection morphisms* $\pi_B^{B\times C} : B \times_{(m,n)} C \to B$ and $\pi_C^{B\times C} : B \times_{(m,n)} C \to C$:

$$\begin{aligned}
\forall s \in S : \left(B \times_{(m,n)} C\right)_s &= \{(x,y) :: m_s(x) = n_s(y)\} \\
\forall f \in O_{w,v} : f^{\left(B\times_{(m,n)}C\right)} &= \left\{(x,y) :: m_f^O(x) = n_f^O(y)\right. \\
\forall f \in O_{w,v} : d_f^{\left(B\times_{(m,n)}C\right)} &::= (x,y) \mapsto d_f^B(x) \,||^w\, d_f^C(y) \\
\forall f \in O_{w,v} : c_f^{\left(B\times_{(m,n)}C\right)} &::= (x,y) \mapsto c_f^B(x) \,||^v\, c_f^C(y),
\end{aligned}$$

where the operator $||^w : B^w \times C^w \to (B \times C)^w$ transforms pairs of w-tuples into w-tuples of pairs: $\left(\left(x_1,\ldots,x_{|w|}\right),\left(y_1,\ldots,y_{|w|}\right)\right) \mapsto \left(\left(x_1,y_1\right),\ldots\left(x_{|w|},y_{|w|}\right)\right)$.

A graph $G \in \mathcal{G}_{\Sigma=(S,O)}$ is a *partial algebra*, if it satisfies the following condition for all $f \in O$:

$$\text{(u)}\ \ \forall e_1, e_2 \in f^G : d_f^G(e_1) = d_f^G(e_2) \implies e_1 = e_2.$$

The full sub-category of $\mathcal{G}_\Sigma$ consisting of all partial algebras[7] is denoted by $\mathcal{A}_\Sigma$ in the following. In a partial algebra A, operation names $f \in O_{\epsilon,v}$ with $|v| > 0$ are interpreted as (partial) *constants*, i. e. $f^A : A^\epsilon \to A^v$ is a partial map from the standard one-element set $A^\epsilon = \{*\}$ into A^v. Symmetrically, operation names $p \in O_{w,\epsilon}$ with $|w| > 0$ are interpreted as *predicates*, since $p^A : A^w \to \{*\}$ is a partial map into the standard one-element set, i. e. it determines a subset on A^w only, namely the part of A^w where it is defined. Finally for operation names $f \in O_{\epsilon,\epsilon}$, there is only two possible interpretations in A, namely $f^A = \emptyset$ (false) or $f^A = \{(*,*)\}$ (true). Thus, f^A is just a boolean flag in this case.

Due to (u) being a set of Horn-axioms, $\mathcal{A}_\Sigma$ is an epireflective sub-category of $\mathcal{G}_\Sigma$, i. e. there is a reflection $\eta : \mathcal{G}_\Sigma \to \mathcal{A}_\Sigma$ that maps a graph $G \in \mathcal{G}_\Sigma$ to

[5] Given a sort indexed family of mappings $(f_s : G_s \to H_s)_{s\in S}$, $f^w : G^w \to H^w$ is recursively defined for every $w \in S^*$ by (i) $f^\epsilon = \{(*,*)\}$, (ii) $f^w = f_s$ if $w = s \in S$, and (iii) $f^w = f^v \times f^u$, if $w = vu$.

[6] The identity morphisms in $\mathcal{G}_\Sigma$ are given by families of identity mappings and composition of morphisms is provided by component-wise composition of the underlying mappings.

[7] Note that the interpretation of an operation name $f \in O_{w,v}$ in a partial algebra A is indeed a partial mapping: due to the uniqueness condition (u), the assignment $\left(f^A, d_f^A : f^A \to A^w, c_f^A : f^A \to A^v\right) \mapsto \left\{\left(d_f^A(e), c_f^A(e)\right) :: e \in f^A\right\}$ provides a partial map from A^w to A^v. And, for a partial map $f : A^w \to A^v$, there is the inverse mapping $f \mapsto \left(f, d_f^A ::= (d,c) \mapsto d, c_f^A ::= (d,c) \mapsto c\right)$ up to renaming of the elements in f^A.

a pair $\left(G^{\mathcal{A}} \in \mathcal{A}_\Sigma, \eta_G : G \to G^{\mathcal{A}}\right)$ such that any graph morphism $h : G \to A$ with $A \in \mathcal{A}_\Sigma$ has a unique extension $h^* : G^{\mathcal{A}} \to A$ with $h^* \circ \eta_G = h$. Since epireflective subcategories are closed wrt. products and sub-objects defined by regular monomorphisms (equalisers), the limits in $\mathcal{A}_\Sigma$ coincide with the limits constructed in $\mathcal{G}_\Sigma$. $\mathcal{A}_\Sigma$ has also all co-limits, since epireflections map co-limits to co-limits. In general, however, the co-limits in $\mathcal{A}_\Sigma$ do not coincide with the co-limits constructed in $\mathcal{G}_\Sigma$. The reflection provides the necessary correction. If, for example, $(b : A \to B, c : A \to C)$ is a span in $\mathcal{A}_\Sigma$ and $(c^* : B \to D, b^* : C \to D)$ is its pushout constructed in $\mathcal{G}_\Sigma$, $\left(\eta_D \circ c^* : B \to D^{\mathcal{A}}, \eta_D \circ b^* : C \to D^{\mathcal{A}}\right)$ is the pushout in $\mathcal{A}_\Sigma$.

Besides being complete and co-complete, the most important property of $\mathcal{A}_\Sigma$ for the rest of the paper is the existence of right adjoints to all inverse image functors. If we fix an algebra $A \in \mathcal{A}_\Sigma$, $\mathcal{A}_\Sigma \downarrow^{\mathrm{M}} A$ denotes the *category of all sub-algebras* of A. The objects in $\mathcal{A}_\Sigma \downarrow^{\mathrm{M}} A$ are all monomorphisms $m : M \rightarrowtail A$ and a morphism in $\mathcal{A}_\Sigma \downarrow^{\mathrm{M}} A$ from $m : M \rightarrowtail A$ to $n : N \rightarrowtail A$ is a (mono)morphism $h : M \rightarrowtail N$ in $\mathcal{A}_\Sigma$ such that $n \circ h = m$. For every $\mathcal{A}_\Sigma$-morphism $g : A \to B$, the *inverse image functor* $g^* : \mathcal{A}_\Sigma \downarrow^{\mathrm{M}} B \to \mathcal{A}_\Sigma \downarrow^{\mathrm{M}} A$ maps an object $m : M \to B \in \mathcal{A}_\Sigma \downarrow^{\mathrm{M}} B$ to $\pi_A^{A \times M} : A \times_{(g,m)} M \to A$ and a morphism $h : (m : M \to B) \to (n : N \to B)$ to the uniquely determined morphism $g^*(h) : A \times_{(g,m)} M \to A \times_{(g,n)} N$ such that $\pi_A^{A \times N} \circ g^*(h) = \pi_A^{A \times M}$ and $\pi_N^{A \times N} \circ g^*(h) = h \circ \pi_M^{A \times M}$.

Fact 1. *In a category $\mathcal{A}_\Sigma$ of partial algebras, every inverse image functor $g^* : \mathcal{A}_\Sigma \downarrow^{\mathrm{M}} B \to \mathcal{A}_\Sigma \downarrow^{\mathrm{M}} A$ has a right adjoint called $g_* : \mathcal{A}_\Sigma \downarrow^{\mathrm{M}} A \to \mathcal{A}_\Sigma \downarrow^{\mathrm{M}} B$.*

Proof. Given a sub-algebra $m : M \rightarrowtail A$, we construct the sub-algebra $g_*(M) \subseteq B$ and the inclusion morphism $g_*(m) : g_*(M) \hookrightarrow B$ as follows:

$$\forall s \in S : g_*(M)_s = \left\{b \in B_s :: \forall\, a \in g_s^{-1}(b)\ \exists\, x \in M : m_s(x) = a \quad \text{and}\right.$$
$$\left.\forall f \in O_{w,v} : f^{g_*(M)} = \left\{e \in f^B :: \forall\, e_a \in g_f^{O\,-1}(e)\ \exists\, e_x \in M : m_f^O(e_x) = e_a\right\}\right.,$$

such that $d_f^{g_*(M)} = d_{f\ |f^{g_*(M)}}^B$ and $c_f^{g_*(M)} = c_{f\ |f^{g_*(M)}}^B$ for every operation symbol.

The co-unit $\varepsilon : g^*\left(g_*\left(m : M \rightarrowtail A\right)\right) \to \left(m : M \rightarrowtail A\right)$ can be defined on every element $(a, b) \in A \times_{(g,g_*(m))} g_*(M)$ by $\varepsilon(a, b) = c$ such that $m(c) = a$. Note that ε is completely defined, since, by definition of $g_*(m)$, a must have a pre-image wrt. m for every pair $(a, b) \in A \times_{(g,g_*(m))} g_*(M)$. It is uniquely defined, since m is monic. By definition of ε, $m \circ \varepsilon = g^*(g_*(m)) = \pi_A^{A \times_{(g,g_*(m))} g_*(M)}$ which means that ε is a morphism in $\mathcal{A}_\Sigma \downarrow^{\mathrm{M}} A$.

Now, let an object $x : X \rightarrowtail B \in \mathcal{A}_\Sigma \downarrow^{\mathrm{M}} B$ and a morphism $k : g^*(x) \to m \in \mathcal{A}_\Sigma \downarrow^{\mathrm{M}} A$, i. e. $m \circ k = \pi_A^{A \times_{(g,x)} X}$ be given. We construct $k^* : x \to g_*(m)$ by $e \mapsto x(e)$ for every $e \in X$. The mappings of k^* are completely defined: (i) if $x(e) \notin g(A)$, $x(e) \in g_*(M)$ because $|g^{-1}(x(e))| = |\left(g \circ m\right)^{-1}(x(e))| = 0$, and, otherwise, the existence of k with $m \circ k = \pi_A^{A \times_{(g,x)} X}$ enforces that every g-pre-image of $x(e)$ has a pre-image under m. By definition, $g_*(m) \circ k^* = x$. By definition of the inverse image functor, $g^*(k^*) : \left(A \times_{(g,x)} X\right) \to \left(A \times_{(g,g_*(m))} g_*(M)\right)$

maps (a, e) to $(a, k^*(e))$. Thus, $\varepsilon\,(g_*(k^*)(a, e)) = \varepsilon(a, k^*(e)) = c$ with $m(c) = a$ and $k(a, e) = c'$ with $m(c') = \pi_A^{A\times(g,x)^X}(a, e) = a$. Since m is monic, $c = c'$. The morphism k^* is uniquely determined, since $g_*(M) \subseteq B$ and $g_*(m)$ is monic. □

3 Partial Morphisms on Partial Algebras

In order to obtain a framework for single-pushout rewriting, we proceed from the category $\mathcal{A}_\Sigma$ of partial algebras with *total* morphisms to the category $\mathcal{A}_\Sigma^{\mathrm{P}}$ of partial algebras and *partial* morphisms. In this section, we investigate the conditions under which pushouts can be constructed in $\mathcal{A}_\Sigma^{\mathrm{P}}$.

A concrete partial morphism over an arbitrary complete category $\mathcal{C}$ is a span of $\mathcal{C}$-morphisms $(p : K \rightarrowtail P, q : K \to Q)$ such that p is monic. Two concrete partial morphisms (p_1, q_1) and (p_2, q_2) are equivalent and denote the same *abstract partial morphism* if there is an isomorphism i such that $p_1 \circ i = p_2$ and $q_1 \circ i = q_2$; in this case we write $(p_1, q_1) \equiv (p_2, q_2)$ and $[(p, q)]_\equiv$ for the class of spans that are equivalent to (p, q). The *category of partial morphisms* $\mathcal{C}^{\mathrm{P}}$ over $\mathcal{C}$ has the same objects as $\mathcal{C}$ and abstract partial morphisms as arrows. The identities are defined by $\mathrm{id}_A^{\mathcal{C}^{\mathrm{P}}} = [(\mathrm{id}_A, \mathrm{id}_A)]_\equiv$ and composition of partial morphisms $[(p : K \rightarrowtail P, q : K \to Q)]_\equiv$ and $[(r : J \rightarrowtail Q, s : J \to R)]_\equiv$ is given by

$$[(r, s)]_\equiv \circ_{\mathcal{C}^{\mathrm{P}}} [(p, q)]_\equiv = [(p \circ r' : M \rightarrowtail P, s \circ q' : M \to R)]_\equiv$$

where $(M, r' : M \rightarrowtail K, q' : M \to J)$ is an arbitrarily chosen but fixed pullback of q and r. Note that there is the faithful embedding functor $\iota : \mathcal{C} \to \mathcal{C}^{\mathrm{P}}$ defined by identity on objects and $(f : A \to B) \mapsto [\mathrm{id}_A : A \rightarrowtail A, f : A \to B]$ on morphisms. We call $[d : A' \rightarrowtail A, f : A' \to B]$ a *total* morphism and, by a slight abuse of notation, write $[d, f] \in \mathcal{C}$, if d is an isomorphism. From now on, we mean the abstract partial morphism $[f, g]_\equiv$ if we write $(f : B \rightarrowtail A, g : B \to C)$.

The single-pushout approach defines direct derivations by a single pushout in a category of partial morphisms. There is a general result for the existence of pushouts in a category $\mathcal{C}^{\mathrm{P}}$ of partial morphisms based on the notions *final triple* and *hereditary pushout* in the underlying category $\mathcal{C}$ of total morphisms.

Definition 2. *(Final triple) A* triple *for a pair* $((l, r), (p, q))$ *of* $\mathcal{C}^{\mathrm{P}}$*-morphisms with common domain is given by a collection* $\left(\overline{p}, p^*, \overline{r}, \overline{l}, l^*, \overline{q}\right)$ *of* $\mathcal{C}$*-morphisms such that* p^**,* $\overline{p}$*,* l^**, and* $\overline{l}$ *are monic and* (i) $(\overline{r}, \overline{p})$ *is pullback of* (r, p^*)*,* (ii) $(\overline{q}, \overline{l})$ *is pullback of* (q, l^*)*, and* (iii) $l \circ \overline{p} = p \circ \overline{l}$*. A triple* $\left(\overline{p}, p^*, \overline{r}, \overline{l}, l^*, \overline{q}\right)$ *for* $((l, r), (p, q))$ *is* final*, if, for any other triple* $\left(p', {p'}^*, r', l', {l'}^*, q'\right)$*, there is a unique collection* (u_1, u_2, u_3) *of* $\mathcal{C}$*-morphisms such that* (iv) $\overline{p} \circ u_1 = p'$*,* (v) $\overline{l} \circ u_1 = l'$*,* (vi) $p^* \circ u_2 = {p'}^*$*,* (vii) $u_2 \circ r' = \overline{r} \circ u_1$*,* (viii) $l^* \circ u_3 = {l'}^*$*, and* (ix) $u_3 \circ q' = \overline{q} \circ u_1$*, compare left part of Fig. 1.*

Definition 3. *(Hereditary pushout) A pushout* (q', p') *of* (p, q) *in* $\mathcal{C}$ *is* hereditary *if for each commutative cube as in the right part of Fig. 1, which has pullback squares* (p_i, i_0) *and* (q_i, i_0) *of* (i_2, p) *and* (i_1, q) *resp. as back faces such that* i_1 *and* i_2 *are monomorphisms, in the top square,* (q_i', p_i') *is pushout of* (p_i, q_i)*, if*

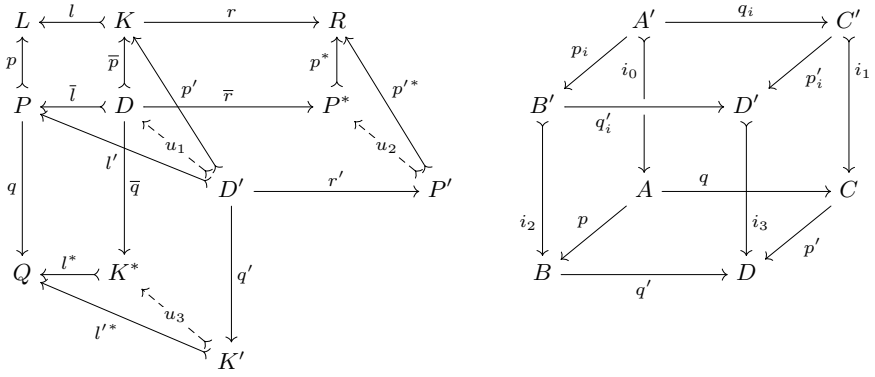


Figure 1. Final Triple and Hereditary Pushout

and only if, in the front faces, (p'_i, i_1) *and* (q'_i, i_2) *are pullbacks of* (i_3, p') *and* (i_3, q') *resp. and* i_3 *is monic.*[8]

Fact 4. *(Pushout in* $\mathcal{C}^{\mathrm{P}}$*) Two partial morphisms* $(l : K \rightarrowtail L, r : K \to R)$ *and* $(p : P \rightarrowtail L, q : P \to Q)$ *have a pushout* $((l^*, r^*), (p^*, q^*))$ *in* $\mathcal{C}^{\mathrm{P}}$*, if and only if there is* (i) *a* final triple $\bar{l} : D \to P$, $\overline{p} : D \to K$, $\overline{r} : D \to P^*$, $\overline{q} : D \to K^*$, $p^* : P^* \to R$, $l^* : K^* \to Q$ *for* $((l, r), (p, q))$ *and* (ii) *a* hereditary pushout $(r^* : K^* \to H, q^* : P^* \to H)$ *for* $(\overline{q}, \overline{r})$ *in* $\mathcal{C}$*, compare sub-diagrams (1) – (3) and (4) resp. in Figure 2.*

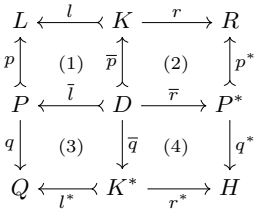


Figure 2. *Pushout in* $\mathcal{G}^{\mathrm{P}}$

The proof can be found in [14]. A version of the proof that does not presuppose a choice of pullbacks that is compatible with pullback composition and decomposition is contained in [15].

Since $\mathcal{A}_\Sigma$ is complete, we can construct the category $\mathcal{A}^{\mathrm{P}}_\Sigma$ of partial algebras and partial morphisms. We use the partial product as the chosen pullback for morphism composition, compare above. The general results about pushouts of partial morphisms carry over to $\mathcal{A}^{\mathrm{P}}_\Sigma$ as follows:

[8] For details on hereditary pushouts see [10,11]

Proposition 5. *(Final triple) Every pair* $((l,r),(p,q))$ *of* $\mathcal{A}_\Sigma^{\mathrm{P}}$*-morphisms with common domain has a final triple.*

Proof. (Sketch) The existence of final triples follows from $\mathcal{A}_\Sigma^{\mathrm{P}}$ being co-complete and having right adjoints for all inverse image functors, compare Fact 1. A detailed proof can be found in [15].

Corollary 6. *(Pushout in* $\mathcal{A}_\Sigma^{\mathrm{P}}$*) A pair of morphisms* $(l : K \rightarrowtail L, r : K \to R)$ *and* $(p : P \rightarrowtail L, q : P \to Q)$ *has a pushout in* $\mathcal{A}_\Sigma^{\mathrm{P}}$*, if and only if the* $\mathcal{A}_\Sigma$*-pushout of* $(\overline{q},\overline{r})$ *is hereditary, where* $\overline{l} : D \to P$*,* $\overline{p} : D \to K$*,* $\overline{r} : D \to P^*$*,* $\overline{q} : D \to K^*$*,* $p^* : P^* \to R$*,* $l^* : K^* \to Q$ *is final triple of* $((l,r),(p,q))$*, see Figure 2.*

Proof. Direct consequence of Fact 4 and Proposition 5.

It is well-known that all pushouts in the category of sets and mappings and in arbitrary categories $\mathcal{G}_\Sigma$ of graphs over a given signature are hereditary. This provides the following sufficient criterion for hereditary pushouts in $\mathcal{A}_\Sigma$.

Proposition 7. *(Sufficient condition) If a pushout in* $\mathcal{A}_\Sigma$ *is also pushout in the larger category* $\mathcal{G}_\Sigma$ *of graphs, then it is hereditary in* $\mathcal{A}_\Sigma$*.*

Proof. Let an arbitrary commutative cube as in the right part of Fig. 1 in $\mathcal{A}_\Sigma$ be given such that the back faces are pullbacks. Then this is also a situation in $\mathcal{G}_\Sigma$ and the back faces are also pullbacks in $\mathcal{G}_\Sigma$, due to $\mathcal{A}_\Sigma$ being an epireflection of $\mathcal{G}_\Sigma$.

Let the front faces be pullbacks in $\mathcal{A}_\Sigma$ and i_3 be a monomorphism. Then the front faces are also pullbacks in $\mathcal{G}_\Sigma$. Since all pushouts in $\mathcal{G}_\Sigma$ are hereditary, D' together with p_i' and q_i' is pushout in $\mathcal{G}_\Sigma$. Since (i) $\mathcal{A}_\Sigma$ is closed wrt. sub-algebras, (ii) D is in $\mathcal{A}_\Sigma$, and (iii) i_3 is monic, D' is also in $\mathcal{A}_\Sigma$ and its reflector $\eta_{D'}$ is an isomorphism. Thus, D' together with p_i' and q_i' is pushout in $\mathcal{A}_\Sigma$.

Let (D', q_i', p_i') be pushout of (p_i, q_i) in $\mathcal{A}_\Sigma$. Construct (D'', q_i'', p_i'') as pushout of (p_i, q_i) in $\mathcal{G}_\Sigma$. We obtain the epic reflector $\eta_{D''} : D'' \twoheadrightarrow D'$ with $p_i' = \eta_{D''} \circ p_i''$ and $q_i' = \eta_{D''} \circ q_i''$. Since D'' is pushout, we also get $i_3' : D'' \rightarrowtail D$ with $i_3' \circ p_i'' = p' \circ i_1$ and $i_3' \circ q_i'' = q' \circ i_2$. Since $i_3 \circ \eta_{D''} \circ p_i'' = i_3 \circ p_i' = p' \circ i_1 = i_3' \circ p_i''$ and $i_3 \circ \eta_{D''} \circ q_i'' = i_3 \circ q_i' = q' \circ i_2 = i_3' \circ q_i''$, we can conclude $i_3 \circ \eta_{D''} = i_3'$. Since all pushouts in $\mathcal{G}_\Sigma$ are hereditary, i_3' is monic implying that $\eta_{D''}$ is monic as well. Thus, $\eta_{D''}$ is an isomorphism and D' is also the pushout in $\mathcal{G}_\Sigma$. This immediately provides monic i_3 and pullbacks in the front faces of the cube in the right part ofFig. 1. □

But not all pushouts in $\mathcal{A}_\Sigma$ are hereditary. Here is a typical example:

Example 8. Consider the signature $\Sigma^c = (S_c, O^c)$ with

$$S_c = \{s\}$$
$$O^c_{w,v} = \begin{cases} \{f\} & w = \epsilon, v = s \\ \emptyset & \text{otherwise,} \end{cases}$$

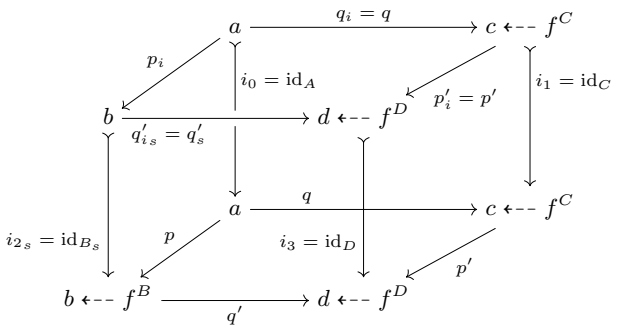


Figure 3. Simple Non-Hereditary Pushout in $\mathcal{A}_\Sigma$

the three algebras

$$\begin{aligned} A ::= A_s = \{a\},\ f^A &= \emptyset, \\ B ::= B_s = \{b\},\ f^B &= \left(\{f^B\},\ d_f^B(f^B) = *,\ c_f^B(f^B) = b\right), \\ C ::= C_s = \{c\},\ f^C &= \left(\{f^C\},\ d_f^C(f^C) = *,\ c_f^C(f^C) = c\right), \end{aligned}$$

and the two morphisms $p : A \to B ::= a \mapsto b$ and $q : A \to C ::= a \mapsto c$.

The pushout of (p, q) in $\mathcal{A}^{\mathrm{P}}_{\Sigma^c}$ consists of the algebra

$$D ::= D_s = \{d\},\ f^D = \left(\{f^D\},\ d_f^D(f^D) = *,\ c_f^D(f^D) = d\right)$$

and the two morphisms

$$\begin{aligned} p' : C \to D ::= c \mapsto d, f^C \mapsto f^D \\ q' : B \to D ::= b \mapsto d, f^B \mapsto f^D. \end{aligned}$$

This pushout is depicted at the bottom of Fig. 3 and is not hereditary. We construct the following cube of morphisms, compare Fig. 3: $A' = A$, $i_0 = \mathrm{id}_A$, B' is defined by $B'_s = B_s$ and $f^{B'} = \emptyset$, i_2 maps b in B'_s to b in B_s, $C' = C$, $i_1 = \mathrm{id}_C$, $q_i = q$, and p_i maps a to b. Note that (i_0, q_i) is pullback of (q, i_1) and (i_0, p_i) is pullback of (p, i_2). Constructing $(D' = D, p'_i = p', q'_i ::= b \mapsto d)$ as the pushout of (p', q'), we obtain $i_3 = \mathrm{id}_D$. But (i_2, q'_i) is not pullback of (q', i_3): $B \times_{(q', i_3)} D'$ contains a defined constant for f, since $i_3(f^D) = q'(f^B)$, and B' does not. □

Note that the $\mathcal{A}_\Sigma$-pushout of the morphisms p and q in Example 8 does not coincide with the pushout of p and q constructed in the larger category $\mathcal{G}_\Sigma$ of graphs. The pushout in $\mathcal{G}_\Sigma$ is the graph

$$G ::= G_s = \{g\}, f^G = \left(\{f_C^G, f_B^G\}, d_f^G(f_C^G) = d_f^G(f_B^G) = *, c_f^G(f_C^G) = c_f^G(f_B^G) = g\right)$$

together with the morphisms

$$\begin{aligned} p'' : C \to G ::= c \mapsto g, f^C \mapsto f_C^G \\ q'' : B \to G ::= b \mapsto g, f^B \mapsto f_B^G. \end{aligned}$$

The partial algebra D is the epireflection of the graph G and the reflector $\eta_G : G \to D$ maps as follows: $g \mapsto d$, $f_C^G \mapsto f^D$, and $f_B^G \mapsto f^D$. The identification $\eta_G(f_C^G) = \eta_G(f_B^G) = f^D$ of the reflector provided the possibility to construct the cube in Example 8 that disproves hereditariness of the pushout of (p, q). The following proposition shows that this construction of a counterexample is always possible if the pushouts in $\mathcal{A}_\Sigma$ and $\mathcal{G}_\Sigma$ are different.

Proposition 9. *(Necessary condition) If a pushout in $\mathcal{A}_\Sigma$ is hereditary, it is also pushout in the larger category $\mathcal{G}_\Sigma$ of graphs.*

Proof. Let $(p : A \to B, q : A \to C)$ be a span of morphisms in $\mathcal{A}_\Sigma$, let $(E, q'' : B \to E, p'' : C \to E)$ be its pushout in $\mathcal{G}_\Sigma$, and let $(q' : B \to D, p' : C \to D)$ be its pushout in $\mathcal{A}_\Sigma$. Since $\mathcal{A}_\Sigma$ is epireflective sub-category of $\mathcal{G}_\Sigma$, we know that $D = E^{\mathcal{A}}$, $q' = \eta_E \circ q''$ and $p' = \eta_E \circ p''$ where $\eta_E : E \to E^{\mathcal{A}}$ is the reflector for the graph E. Suppose D and E are not isomorphic, then there are $e_1 \neq e_2$ with $\eta_E(e_1) = \eta_E(e_2)$. We distinguish two cases: $e_1, e_2 \in E_s$ for some sort $s \in S$ and $e_1, e_2 \in f^E$ for some operation name $f \in O_{w,v}$ and $w, v \in S^*$.

In the first case, construct the following commutative cube, compare right part of Fig. 1: A', B', and C' have the same carrier sets as A, B, and C respectively, their operations, however, are completely undefined. The embeddings i_0, i_1, and i_2 are identities on the carriers and empty mappings on the operations. The morphisms q_i and p_i coincide with q and p respectively on the carriers and are empty for all operations. Note that (q_i, i_0) and (p_i, i_0) are pullbacks of (q, i_1) and (p, i_2) respectively. Construct $(q_i' : B' \to D', p_i' : C' \to D')$ as the pushout of (p_i, q_i) in $\mathcal{G}_\Sigma$. Since all operations are undefined, it is also pushout in $\mathcal{A}_\Sigma$. And we know, that $D_s' = E_s$ for all sorts $s \in S$. Thus, $e_1 \neq e_2$ in D' and i_3 is not monic.

In the second case, we can, without loss of generality, suppose $E_s = D_s$ for all sorts $s \in S$. Since p' and q' are jointly epic, both e_1 and e_2 have pre-images under p' and/or q'. Let $e_1', e_2' \in f^B \uplus f^C$ be those pre-images and suppose, without loss of generality, $e_1' \in f^B$. Since $e_1 \neq e_2$, we conclude $[e_1']_{\equiv^f} \neq [e_2']_{\equiv^f}$, where the equivalence $\equiv^f \subseteq \left(f^B \uplus f^C\right) \times \left(f^B \uplus f^C\right)$ is generated by $\left\{ \left(p_f^O(e), q_f^O(e)\right) :: e \in f^A \right\}$. Construct the following cube à la Fig. 1(right part): The algebras A', B', and C' coincide in all carriers and operations except f with A, B, and C respectively. For f, we let

$$
\begin{aligned}
f^{B'} &= f^B - \{e \in [e_1']_{\equiv^f} :: e \in f^B\} \\
f^{C'} &= f^C - \{e \in [e_1']_{\equiv^f} :: e \in f^C\} \\
f^{A'} &= f^A - \{e \in f^A :: q_f^O(e) \in [e_1']_{\equiv^f} \vee p_f^O(e) \in [e_1']_{\equiv^f}\}.
\end{aligned}
$$

By this construction, we erase the whole structure that generated $[e_1']_{\equiv^f}$ from A, B, and C. Note that, due to $[e_1']_{\equiv^f} \neq [e_2']_{\equiv^f}$, e_2' is kept in f^B or f^C. Let i_0, i_1, and i_2 be the natural inclusions. And let q_i and p_i be the restrictions of q and p to A'. Since we erased the whole equivalence class $[e_1']_{\equiv^f}$, (q_i, i_0) and (p_i, i_0) are pullbacks of (q, i_1) and (p, i_2) respectively. Let (D', q_i', p_i') be the pushout of (q_i, p_i). Then, (i_2, q_i') is not pullback of (i_3, q'): By assumption,

$q'(e'_1) = e_1 = i_3(x)$ where $x = q'_i(e'_2)$ or $x = p'_i(e'_2)$. The function entry e'_1, however, does not possess a pre-image under i_2. □

Theorem 10. *A pushout in $\mathcal{A}_\Sigma$ is hereditary, if and only if it is pushout in $\mathcal{G}_\Sigma$.*

Proof. Direct consequence of Propositions 7 and 9.

Corollary 11. *Morphisms $(l : K \rightarrowtail L, r : K \to R)$ and $(p : P \rightarrowtail L, q : P \to Q)$ have a pushout in $\mathcal{A}^{\mathrm{P}}_\Sigma$, if and only if the $\mathcal{A}_\Sigma$-pushout of $(\overline{q}, \overline{r})$ is pushout in $\mathcal{G}_\Sigma$, where $\left(\overline{l}, \overline{p}, \overline{r}, \overline{q}, p^*, l^*\right)$ is final triple for $((l, r), (p, q))$, compare Figure 2.*

4 Single-Pushout Rewriting of Partial Algebras

In this section, we introduce single-pushout rewriting of partial algebras. We restrict rules to partial morphisms $(l : K \rightarrowtail L, r : K \rightarrowtail R)$ that do not identify items, i. e. the right-hand side of which are injective. Furthermore, we only allow matches that produce total co-matches. For this set-up, we can characterise the application conditions stipulated by the absence of some pushouts in categories of partial algebras with partial morphisms. And we can show a close connection of single-pushout and sesqui-pushout rewriting. In the following, let $\mathcal{A}^{\mathrm{P}}_\Sigma$ be a category of partial algebras and partial morphisms with respect to a given signature $\Sigma = \left(S, (O_{w,v})_{w,v \in S^*}\right)$.

Definition 12. *(Rule, match, and transformation) A* transformation rule *t is a* partial *morphism $t = (l : K \rightarrowtail L, r : K \rightarrowtail R)$ the right-hand side r of which is injective. A* match *for a rule $t : L \to R$ in a host algebra G is a* total *morphism $m : L \to G$. A* direct transformation *with a rule $t : L \to R$ at a match $m : L \to G$ from algebra G to algebra $t@m$ exists if there is a* total co-match *$m\langle t \rangle : R \to t@m$ and a partial* trace *$t\langle m \rangle : G \to t@m$, such that $(t\langle m \rangle, m\langle t \rangle)$ is pushout of t and m in $\mathcal{A}^{\mathrm{P}}_\Sigma$.*

There are two reasons why a transformation with a rule r at a match m cannot be performed: (i) There is no pushout of t and m in $\mathcal{A}^{\mathrm{P}}_\Sigma$ and (ii) the co-match in the pushout of t and m is not total. Therefore, we have some application conditions as in the double-pushout approach [5]. Since we restricted the rules to right-hand sides which do not identify any items, the application conditions can easily be characterised.[9]

Proposition 13. *(Application conditions) A transformation with a rule $t : L \to R$ at a match $m : L \to G$ exists, if and only if*

1. *the match does not identify items that are preserved with items that are deleted by the rule, i. e. for all $x \neq y \in L : m(x) = m(y)$ and t defined for x implies that t is also defined for y,*

[9] Note that Definition 12 can be generalised to arbitrary right-hand sides in rules. In the general case, however, the application condition introduced by the requirement that participating pushouts are hereditary is more complex.

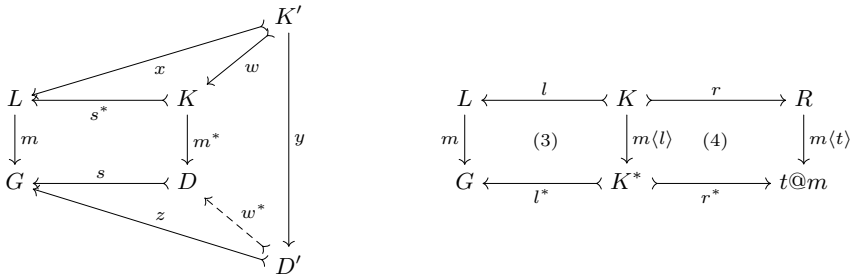


Figure 4. Single- versus Sesqui-Pushout Transformation

2. *the rule does not add operation definitions that are already present in the host graph G, i. e. for all $w, v \in S^*, f \in O_{w,v}, x \in L^w, e_R \in f^R, e_G \in f^G$:*

$$d_f^R(e_R) = t^w(x) \wedge d_f^G(e_G) = m^w(x) \implies \exists e_L \in f^L : m(e_L) = e_G,$$

3. *and the match does not identify domains of different added operation definitions, i. e. for all $w, v \in S^*, f \in O_{w,v}, e_1 \neq e_2 \in f^R$:*

$$m\langle t\rangle^w\left(d_f^R(e_1)\right) = m\langle t\rangle^w\left(d_f^R(e_2)\right) \implies \exists e_1' \in f^L : e_1 = t_f^O(e_1').$$

Note that the second clause above also implies $t(e_L) = e_R$ and the third clause also implies $\exists e_2' : e_2 = t_f^O(e_2')$.

Proof. The first condition is the well-known condition which is called *conflict-free* in [12] that characterises matches that produce pushouts in $\mathcal{G}_\Sigma$ with total co-match. Conditions 2 and 3 translate the result of Corollary 11 to the concrete situation where r is monic and p, $\overline{p}$, and p^* are isomorphisms. □

Since we restricted transformations to total co-matches, we obtain a close connection of our transformations to Sesqui-Pushout Rewritings in the sense of [3], which are composed of final pullback complements and pushouts.

Definition 14. *(Final Pullback Complement) In a pullback (s^*, m^*) of (m, s), compare left part of Fig. 4, the pair (s, m^*) constitutes a final pullback complement of (m, s^*), if for any other pullback (x, y) of (m, z) and morphism w such that $s^* \circ w = x$ there is a unique morphism w^* with $s \circ w^* = z$ and $w^* \circ y = m^* \circ w$.*

Theorem 15. *(Single- and Sesqui-Pushout Transformation) Given a rule $t = (l : K \rightarrowtail L, r : K \rightarrowtail R)$, a match $m : L \to G$, and a direct transformation $(m\langle t\rangle, t\langle m\rangle = (l^* : K^* \rightarrowtail G, r^* : K^* \rightarrowtail t@m))$, then there is a total morphism $m\langle l\rangle : K \to K^*$ such that $(l^*, m\langle l\rangle)$* and *$(r^*, m\langle l\rangle)$* are final pullback complements of (m, l) and $(m\langle t\rangle, r)$ resp., compare (3) and (4) in Fig. 4.

Proof. That $(l^*, m\langle l\rangle)$ is final pullback complement of (m, l) is a direct consequence of the construction of final triples in [15] and the fact that the co-match

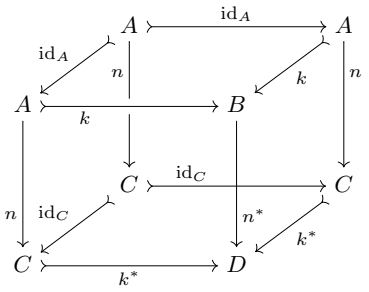


Figure 5. Hereditary Pushout and Final Pullback Complement

is total. It remains to show that hereditary pushouts along monomorphisms are final pullback complements as well: Let (k^*, n^*) be hereditary pushout of n and monic k. We construct the commutative cube in Fig. 5, in which the morphisms k, id_C, and id_A are monic, the top and left face are pullbacks and the back face is pushout. By (k^*, n^*) being hereditary, the bottom and the right face are pullbacks and k^* is monic. Moreover in the left face, (id_C, n) is final pullback complement of (n, id_A) and the back face is hereditary. Thus, the left and the back face constitute a pushout in the category of partial morphisms. The front face is hereditary and, therefore, pushout in the category of partial morphisms. Since $(\mathrm{id}_A, \mathrm{id}_A) = (k, \mathrm{id}_A) \circ (\mathrm{id}_A, k)$ and $(\mathrm{id}_C, \mathrm{id}_C) = (k^*, \mathrm{id}_C) \circ (\mathrm{id}_C, k^*)$, the right face must be pushout in the category of partial morphisms and (k^*, n) must be final pullback complement of (n^*, k). □

Thus, single-pushout rewriting with right-linear rules and total co-matches is almost Sesqui-Pushout Rewriting. This analysis provides good chances to reestablish most of the theory known for the single- and the sesqui-pushout approach, for example with respect to parallel and sequential independence, concurrency, and amalgamation. And it shows that our approach is closely connected to some other current research lines, for example [4]. But the application conditions for transformations in $\mathcal{A}^{\mathrm{P}}_{\Sigma}$ in Proposition 13 also produce some *new* and *unfamiliar* behaviour, for example if decomposition of rules is concerned.

Example 16. (Transformation Decomposition) In the standard single-pushout approach at injective matches, rule decomposition carries over to transformations: If a rule t can be decomposed into two rules t_1 and t_2, i. e. $t = t_2 \circ t_1$, every transformation with rule t at an injective match m can be decomposed into a transformation with t_1 followed by a transformation with t_2, such that $t\langle m \rangle = t_2 \langle m \langle t_1 \rangle \rangle \circ t_1 \langle m \rangle$ and $m \langle t \rangle = (m \langle t_1 \rangle) \langle t_2 \rangle$. This is no longer true in the new set-up. Consider again the signature of Example 8, the partial algebras

$$
\begin{aligned}
L ::= L_s &= \{l\},\ f^L = \emptyset, \\
E ::= E_s &= \{e\},\ f^E = \left(\{f^E\},\ d^E_f(f^E) = *,\ c^E_f(f^E) = e\right), \\
R ::= R_s &= \{r\},\ f^R = \emptyset, \text{ and} \\
G ::= G_s &= \{g\},\ f^G = \left(\{f^G\},\ d^G_f(f^G) = *,\ c^G_f(f^G) = g\right),
\end{aligned}
$$

the rules $t_1 : L \to E ::= l \mapsto e$ and $t_2 : E \to R ::= e \mapsto r$, and the match $m : L \to G ::= l \mapsto g$. Note that t_2 is partial, since it does not map the operation definition in E. Since $t_2 \circ t_1 : L \to R$ is a rule without new operation definitions in R, there is the transformation $(t_2 \circ t_1)\,@m$. The rule t_1, however, cannot be applied at m due to a violation of the second condition in Proposition 13. □

The careful analysis of these new features is left to future research.

5 Examples

The new behaviour discovered in Example 16, can be usefully exploited in many practical applications as a condition that prevents rule application. Our first

Figure 6. Setting and Changing an Attribute

example is a simple integer attribute `i` that can be set or changed for objects of type `O`. Figure 6 shows the underlying signature[10] and the two rules. Note that the `set`-rule can only be applied in a situation where the `i`-attribute of `o` has not been set yet, compare the second condition in Proposition 13. If there is an old value, the `change`-rule must be applied.

Figure 7. Reflexive/Transitive Closure

The next example handles the reflexive and transitive closure of a relation on the set `O`. We just apply the two rules `reflexive` and `transitive` as long as there are matches. Note that the algorithm terminates, since the rule `reflexive` cannot add loops to objects that possess a loop already, compare the second condition in Proposition 13. If all abbreviations are added, also the rule `transitive` is not applicable any more.

The last example shows a typical copying process, here for a tree structure, compare Figure 8. The partial dyadic operation `t` builds up trees, the unary predicate `r` marks the root for the copy process, the `start` and the three `copy` rules perform the copy process, and the operation `c` keeps track of already built copies. Again, the application conditions of single pushout rewriting for partial

[10] In the signature, we declare the visualisations for the operations in brackets.

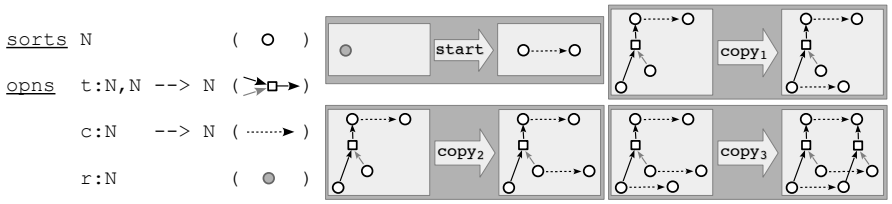


Figure 8. Copying

algebras guarantee that *exactly one* copy is made. Note that this copy mechanism also works if `t` builds up arbitrary hierarchical or even cyclic structures.

This last example describes a typical software engineering situation in the area of model transformation: some structured model, for example a class model, has to be transformed into another structured system, for example a relational database. In this context, keeping track of already finished transformations is essential for the control flow of the transformation process and to avoid that some parts are performed twice.

More detailed examples in this area can be found in [15].

6 Related Work and Conclusions

We have introduced single-pushout rewriting of arbitrary partial algebras. As usual, transformations are defined by a single pushout of partial morphisms. Thus, general composition and decomposition properties of pushouts can be exploited for a rich theory. The new approach is built on a category of partial morphisms that *does not have all* pushouts. We provided a good characterisation of the situations which admit pushouts by hereditariness of underlying pushouts of total morphisms, compare Theorem 10. Informally, pushouts can be built if the applied rule does not try to define operations where they are defined already. This application condition can easily be checked in every concrete situation. By some examples, we showed the practical relevance of the application conditions for system design and the termination of derivation sequences. (More examples can be found in [15].) Within our approach, we do not have to distinguish between graph structures (objects and links) and data structures (base-types and -operations). We can easily model associations and attributes with at-most-one-multiplicity.

There are only a few articles in the literature that address rewriting of partial algebras, for example [2] and [1] for the double- and single-pushout approach resp. But both papers stay in the framework of signatures with *unary* operation symbols only and aim at an underlying category that is co-complete.

Aspects of partial algebras occur in all papers that are concerned with relabelling of nodes and edges, for example [9], or that invent mechanisms for exchanging the attribute value without deleting and adding an object, for example [7]. Most of these approaches avoid "real" partial algebras by completing them to total ones by some undefined-values.

Thus, our approach is new, shows some application potentials, and seems promising wrt. theoretical results.

References

1. Peter Burmeister, Miquel Monserrat, Francesc Rosselló, and Gabriel Valiente. Algebraic transformation of unary partial algebras II: single-pushout approach. *Theor. Comput. Sci.*, 216(1-2):311–362, 1999.
2. Peter Burmeister, Francesc Rosselló, Joan Torrens, and Gabriel Valiente. Algebraic transformation of unary partial algebras I: double-pushout approach. *Theor. Comput. Sci.*, 184(1-2):145–193, 1997.
3. Andrea Corradini, Tobias Heindel, Frank Hermann, and Barbara König. Sesqui-pushout rewriting. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *ICGT*, volume 4178 of *Lecture Notes in Computer Science*, pages 30–45. Springer, 2006.
4. Vincent Danos, Tobias Heindel, Ricardo Honorato-Zimmer, and Sandro Stucki. Reversible sesqui-pushout rewriting. In *Graph Transformation - 7th International Conference, ICGT 2014, Held as Part of STAF 2014, York, UK, July 22-24, 2014. Proceedings*, pages 161–176, 2014.
5. Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.
6. Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Graph Transformations - 6th International Conference, ICGT 2012, Bremen, Germany, September 24-29, 2012. Proceedings*, volume 7562 of *Lecture Notes in Computer Science*. Springer, 2012.
7. Ulrike Golas. A general attribution concept for models in M-adhesive transformation systems. In Ehrig et al. [6], pages 187–202.
8. Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundam. Inform.*, 26(3/4):287–313, 1996.
9. Annegret Habel and Detlef Plump. M,n-adhesive transformation systems. In Ehrig et al. [6], pages 218–233.
10. Tobias Heindel. Hereditary pushouts reconsidered. In Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr, editors, *ICGT*, volume 6372 of *Lecture Notes in Computer Science*, pages 250–265. Springer, 2010.
11. Richard Kennaway. Graph rewriting in some categories of partial morphisms. In Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 490–504. Springer, 1990.
12. Michael Löwe. Algebraic approach to single-pushout graph transformation. *Theor. Comput. Sci.*, 109(1&2):181–224, 1993.
13. Michael Löwe, Martin Korff, and Annika Wagner. An algebraic framework for the transformation of attributed graphs. In M. R. Sleep et al, editor, *Term Graph Rewriting: Theory and Practice*, pages 185 – 199. Wiley, 1993.
14. Miquel Monserrat, Francesc Rossello, Joan Torrens, and Gabriel Valiente. Single pushout rewriting in categories of spans I: The general setting. Technical Report LSI-97-23-R, Department de Llenguatges i Sistemes Informtics, Universitat Politcnica de Catalunya, 1997.
15. Marius Tempelmeier and Michael Löwe. Single-Pushout Transformation partieller Algebren. Technical Report 2015/1 (in German), FHDW-Hannover, 2015.

On Correctness of Graph Programs Relative to Recursively Nested Conditions

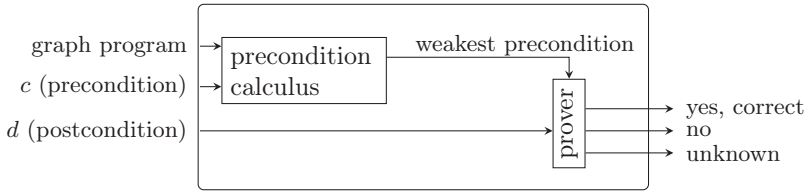
Nils Erik Flick*

Carl von Ossietzky Universität, 26111 Oldenburg, Germany,
flick@informatik.uni-oldenburg.de

Abstract. We propose a new specification language for the proof-based approach to verification of graph programs by introducing μ-conditions as an alternative to existing formalisms which can express path properties. The contributions of this paper are the lifting of constructions from nested conditions to the new, more expressive conditions, and a proof calculus for partial correctness relative to μ-conditions.

1 Introduction

Graph transformations provide a formal way to model the graph-based behaviour of a wide range of systems by way of diagrams. Such systems can be formally verified. One approach to verification proceeds via model checking of abstractions, notably Gadducci et al., Baldan et al., König et al., Rensink et al. [4, 1, 10, 18]. This can be contrasted with the proof-based approaches of Habel, Pennemann and Rensink [7, 6] and Poskitt and Plump [15]. Here, state properties are expressed by nested graph conditions, and a program can be proved correct with respect to a precondition c and a postcondition d. The following figure presents a schematic overview of the approach, which is also our starting point:



The correctness proof is done in the style of Dijkstra's [2] predicate transformer approach in Pennemann's thesis [12], while Poskitt's thesis [14] features a Hoare [8] logic for partial and total correctness. Both works are based on nested conditions, which cannot express non-local properties of graphs, such as connectivity. In this paper, we consider non-local properties, and we present an extension to the proof calculus from [12].

* This work is supported by the German Research Foundation (DFG), grant GRK 1765 (Research Training Group – System Correctness under Adverse Conditions)

Our formalism is an extension of nested conditions by recursive definitions. While several extensions of nested graph conditions to non-local conditions already exist (Radke [17], Poskitt and Plump [16]), we argue that as opposed to the former, ours already offers a weakest precondition calculus that can handle any condition expressible in it; as compared to the latter, which relies more heavily on expressing properties directly in (monadic second-order) logic, ours is more closely related to nested conditions and shares the same basic methodology. Therefore μ-conditions, albeit still work in progress, offer a new viewpoint that may be sufficiently different from existing ones to be worth investigating.

The outline of the paper is as follows: Section 2 recalls graph programs and conditions. Sections 3 and 4 introduce μ-conditions and correctness under μ-conditions, respectively, together with an exemplary application of the method, Section 5 provides context by listing related work and Section 6 concludes with an outlook. After the main text, there is an appendix with the proofs.

2 Graph Conditions and Programs

In this section, we introduce graph conditions and graph programs. We assume familiarity with graph transformation systems in the sense of Ehrig et al. [3], and the basic notions of category theory. For standard definitions and more details, we refer the reader to Ehrig et al. [3]. For an in-depth introduction to nested conditions and graph programs and practical approaches to semi-automatic theorem proving in this context, we refer the reader to Pennemann [12].

Notation. The domain and codomain of a morphism $f : G \to H$ are denoted by $\mathrm{dom}(f) = G$ and $\mathrm{cod}(f) = H$. Injective morphisms (monomorphisms) are distinguished typographically by a curly arrow $f : G \hookrightarrow H$ while double-tailed arrows $f : G \twoheadrightarrow H$ denote surjective ones (epimorphisms). We use the symbol $\mathcal{M}$ to denote the class of all graph monomorphisms. A *partial morphism* is a pair of monomorphisms with the same domain. The empty graph is denoted by $\emptyset$.

All graphs in this paper are assumed to be finite.

A brief review of nested conditions follows. Nested graph conditions were proposed by Habel and Pennemann. Finite nested conditions were later shown to be equally expressive as graph-interpreted first-order predicate logic. Graph conditions can be used as constraints to specify state properties, or as application conditions to restrict the applicability of a rule.

Definition 1 (Nested Graph Conditions). Let Cond be the class of nested conditions, defined inductively as follows (where P, C', C are graphs):

- If J is a countable set and for all $j \in J$, c_j is a condition (over P), then $\bigvee_{j \in J} c_j$ is a condition (over P). This includes the case $J = \emptyset$ (for any P).
- If c is a condition (over P), then $\neg c$ is also a condition (over P).

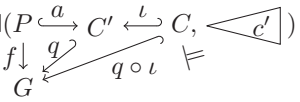
– If $a : P \hookrightarrow C'$ is a monomorphism, $\iota : C \hookrightarrow C'$ is a monomorphism and c' is a condition (over C), then $\exists(a, \iota, c')$ is a condition (over P).

We call c' a *direct subcondition* of $\exists(\alpha, \iota, c')$, $\neg c'$ and $c' \vee c''$ and use *subcondition* for the reflexive and transitive closure of this syntactically defined relation.

Notation. If c is a condition over P, then P is its *type*[1] and we write $c : P$, and Cond_P is the class of all conditions over P. We may write $\exists(a, c)$ instead of $\exists(a, id_{\mathrm{cod}(a)}, c)$. The usual abbreviations define the other standard operators: $\bigwedge$ is $\neg \bigvee \neg$, $\forall$ is $\neg\exists\neg$. No morphism satisfies the disjunction over the empty index set. We introduce *false* as a notation for it, and *true* for $\neg$*false*. We may omit the subcondition *true* (together with ι), writing $\exists(a)$ for $\exists(a, \iota, \mathit{true})$.

When all the index sets are finite, one obtains the *finite* nested conditions. The morphism ι serves to unselect[2] a part of C', which will become necessary later.

Definition 2 (Satisfaction). A monomorphism $f : P \hookrightarrow G$ *satisfies* a condition $c : P$, denoted $f \models c$, iff $c = \mathit{true}$, $c = \neg c'$ and $f \not\models c'$, or $c = \bigvee_{j \in J} c_j$ and there is a $j \in J$ such that $f \models c_j$, or $c = \exists(a, \iota, c')$ ($a : P \hookrightarrow C'$, $\iota : C \hookrightarrow C'$, $c' : C$) and there exists a monomorphism $q : C' \hookrightarrow G$ such that $f = q \circ a$ and $q \circ \iota \models c'$.



A graph G satisfies a condition $c : \emptyset$ iff the unique morphism $\emptyset \hookrightarrow G$ satisfies c.

In the diagram of Def. 2, the triangle indicates that C is the type of the subcondition c' which appears nested inside $\exists(\alpha, \iota, c')$.

Remark 1 (No Added Expressivity)**.** Our conditions with ι are equally expressive as the nested conditions defined in [12]. The proof, which we omit here, relies on the transformation A from [12].

Definition 3. $\equiv$ denotes logical equivalence, i.e. for conditions $c, c' : P$, $c \equiv c'$ iff for all monomorphisms m with domain P, $\Rightarrow m \models c \Leftrightarrow m \models c'$.

Notation. As one can see in Fig. 1, the notation for graph conditions customarily only depicts source or target graphs of morphisms. The tiny blue numbers

[1] when we mention "type graphs" in the text, we just mean graphs used as types.

[2] We will use the term "unselection" anytime a morphism is used in the inverse direction: in Def. 1, the morphism ι is used to base subconditions on a smaller subgraph, in effect reducing the *selected* subgraph; it will also appear in our definition of graph programs as the name of an operation that reduces the current *selection*, i.e. the subgraph the program is currently working on – similarly for "selection".

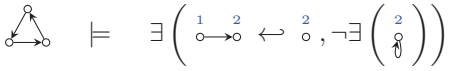


Fig. 1. A nested graph condition (stating the existence of two nodes linked by an edge, where the second node does not have a self-loop) and a graph satisfying it.

show the morphisms' node mappings. We also adopt the convention of not explicitly representing the morphism ι in a situation $\exists(a, \iota, \mathtt{x}_i)$; we prefer to annotate the variable's type graph with the images of items under ι in parentheses.

Next, we introduce graph transformations. We follow the double pushout approach with injective rules and injective matches. For technical reasons, we define graph transformations in terms of four elementary steps, namely selection, deletion, addition and unselection. Deletion and addition always apply to a selected subgraph, and selection and unselection allow the selection to be changed. *skip* is a no-op used in the definition of sequential composition. The definition below allows for somewhat more general combinations of the basic steps, which cannot be expressed by sets of graph transformation rules.

The semantics of a graph program is a triple of two monomorphisms and one partial morphism. The two monomorphisms represent the selected subgraphs before and after the execution of the program respectively, and the partial morphism records the changes effected by the program. Our programs are a proper subset of those in Pennemann [12], and use the same semantics.

Definition 4 (Graph Programs).

In the following table, x, l, r, y, m_{in} and m_{out} are monomorphisms, with x, l, r and y arbitrarily chosen to define a program step, while m_{in} and m_{out} are called *interfaces* and universally quantified in the set comprehensions that appear in the definitions below.

Name	Program P	Semantics $[\![P]\!]$
selection	$Sel(x)$	$\{(m_{in}, m_{out}, x) \mid m_{out} \circ x = m_{in}\}$
deletion	$Del(l)$	$\{(m_{in}, m_{out}, l^{-1}) \mid \exists l', (m_{out}, l, m_{in}, l') \text{ pushout}\}$
addition	$Add(r)$	$\{(m_{in}, m_{out}, r) \mid \exists r', (m_{in}, r, m_{out}, r') \text{ pushout}\}$
unselection	$Uns(y)$	$\{(m_{in}, m_{out}, y^{-1}) \mid m_{out} = m_{in} \circ y\}$
skip	$skip$	$\{(m, m, id_{\mathrm{dom}(m)}) \mid m \in \mathcal{M}\}$

If P and Q are graph programs, then so are their disjunctive $\{P, Q\}$ and sequential $(P; Q)$ composition. The semantics of disjunction is a set union $[\![P]\!] \cup [\![Q]\!]$ and the semantics of sequence is $[\![P; Q]\!] = \{(m, m', p) \mid \exists (m, m'', p') \in [\![P]\!], (m'', m', p'') \in [\![Q]\!], p = p'; p''\}$, where composition $p'; p''$ of partial morphisms $p' = G_1 \overset{l_1}{\hookleftarrow} D_1 \overset{r_1}{\hookrightarrow} H_1$, $p'' = H_1 \overset{l_2}{\hookleftarrow} D_2' \overset{r_2}{\hookrightarrow} H_2$ is defined as $G \overset{l_1 \circ l_2'}{\hookleftarrow} D'' \overset{r_2 \circ r_1'}{\hookrightarrow} H_2$ using the pullback (r_1', l_2') of (r_1, l_2). If P is a graph program, then so is its *iteration* P^*; $[\![P^*]\!] = \bigcup_{j \in \mathbb{N}} [\![P^j]\!]$ where $P^j = P; P^{j-1}$ for $j \geq 1$ and $P^0 = skip$.

Remark 2. The definitions generalise the state transitions in plain graph transformation, a rule $\varrho = (L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R)$ being precisely simulated by the program $Sel(\emptyset \hookrightarrow L); Del(l); Add(r); Uns(\emptyset \hookrightarrow R)$.

3 μ-Conditions

In this section, we define μ-conditions on the basis of nested graph conditions. These are capable of expressing path and connectivity properties, which frequently arise in the study of the correctness of programs with recursive data structures, or in the modelling of networks. We then define and prove the correctness of some basic constructions. An example is provided at the end of this section to illustrate the constructions step by step.

3.1 Defining μ-Conditions

Nested conditions are a very successful approach to the specification of graph properties for verification. However, they are unable to express non-local properties such as connectedness. Our idea is to generalise nested conditions to capture certain non-local properties by adding recursion. The resulting formalism will be similar to first order fixed point logics, see e.g. Kreutzer [9]. The reader might want to compare our μ-conditions to a distinct formalism towards expressing non-local properties, the very powerful grammar-based HR* conditions of Radke [17]. We argue that μ-conditions are worth looking into despite the availability of strong contenders for the extension of nested conditions to non-local properties, such as MSO-conditions [16] because μ-conditions provide a new and different generalisation of nested conditions, and neither is it immediately clear how the respective expressivities compare. The related work section, Sec. 5, contains a summary on different non-local graph condition formalisms. Specifically, we will show in this section that the weakest liberal precondition transformation, core of the Dijkstra-style approach, can be adapted.

Notation. Sequences (of graphs, placeholders, morphisms) are written as bold letters $\boldsymbol{P}$, $\mathtt{x}$, $\boldsymbol{f}$, and their components are numbered starting from 1. The length of a sequence $\boldsymbol{P}$ is denoted by $\|P\|$. Indexed typewriter letters $\mathtt{x}_1$ stand for placeholders, i.e. variables. The notation $c : P$ indicating that c has type P is also extended to sequences: $\boldsymbol{c} : \boldsymbol{P}$ (provided $\|c\| = \|P\|$).

To define fixed point conditions, we need something to take fixed points of, and to ascertain that the fixed point exists. Choosing a partial order on $\mathrm{Cond}_{\boldsymbol{P}}$, one can define monotonic operators on $\mathrm{Cond}_{\boldsymbol{P}}$. The semantics of satisfaction already defines a pre-order: $c \leq c'$ iff every morphism that satisfies c also satisfies c', which is obviously transitive and reflexive. As in every pre-order, $\leq \cap \leq^{-1}$ is an equivalence relation compatible with $\leq$ and comparing representants via $\leq$ partially orders its equivalence classes. We introduce variables as placeholders where further conditions can be substituted[3].

[3] Note that in our approach variables stand only for subconditions, not for attributes or parts of graphs. Wherever confusion with similarly named concepts from the literature could arise, we will use the word "placeholder" meaning "variable".

To represent systems of simultaneous equations, we work on tuples of conditions. If $\boldsymbol{P} = P_1, \ldots, P_{\|P\|}$ is a sequence of graphs, then $\mathrm{Cond}_{\boldsymbol{P}}$ is the set of all $\|P\|$-tuples $\boldsymbol{c}$ of conditions, whose i-th element is a condition over the i-th graph of $\boldsymbol{P}$. Satisfaction is defined component-wise: $\boldsymbol{f} \models \boldsymbol{c}$ iff $\forall k \in \{1, \ldots, \|P\|\}\ f_k \models c_k$.

By definition, $\bigwedge$ and $\bigvee$ of countable sets of Cond_P conditions exist for any P, and they are easily seen to be least upper and greatest lower bounds of the sets. This makes $\mathrm{Cond}_{\equiv P}$ a complete lattice. Let $\mathrm{Cond}_{\boldsymbol{P}}$ be ordered with the product order by defining $\boldsymbol{f} \models \boldsymbol{c}$ to be true when the conjunction holds. This again induces a partial order on the set of equivalence classes, $\mathrm{Cond}_{\equiv \boldsymbol{P}}$. Thus, $\mathrm{Cond}_{\equiv \boldsymbol{P}}$ is also a complete lattice, and monotonic operators have least fixed points by the Knaster-Tarski theorem [20], given by the limit of $\boldsymbol{\mathcal{F}}^n(\boldsymbol{false})$ for all $n \in \mathcal{N}$. This ensures that systems of equations as defined below yield least fixed point solutions, which is crucial in the definition of a μ-condition. We extend the inductive definition from Def. 1 as follows:

Definition 5 (Graph Conditions with Placeholders). Given a graph P and a finite sequence $\boldsymbol{P}$ of graphs or morphisms, a *condition with placeholders from* $\boldsymbol{P}$ over P is a (graph) condition with placeholders is either $\exists(a, \iota, c)$, or $\neg c$, or $\bigvee_{j \in J} c_j$, or $\mathtt{x}_i$, $1 \leq i \leq \|\boldsymbol{P}\|$ where $\mathtt{x}_i$ is a variable of type P_i.

A condition can be substituted for a variable of same type:

Definition 6 (Substitution). If $\boldsymbol{P}$ is a list of graphs and $\mathcal{F}$ is a condition with placeholders $\mathtt{x}$ over $\boldsymbol{P}$, then if $\boldsymbol{c} \in \mathit{Cond}_{\boldsymbol{P}}$, then $\mathcal{F}[\mathtt{x}/\boldsymbol{c}]$ is obtained by substituting each occurrence of $\mathtt{x}_i$ by c_i for all $i \in \{1, ..., \|\mathcal{P}\|\}$.

Satisfaction of conditions with placeholders by a morphism f is defined in the obvious way relative to a *valuation*, which is an assignment of *true* or *false* to each monomorphism of the type graph of the variable into $\mathrm{cod}(f)$.

As discussed above, a least fixed point will be sought only up to logical equivalence. To guarantee existence of the least fixed point, the operator must be monotonic ($\boldsymbol{c} \leq \boldsymbol{d} \Rightarrow \boldsymbol{\mathcal{F}}(\boldsymbol{c}) \leq \boldsymbol{\mathcal{F}}(\boldsymbol{d})$ for any $\boldsymbol{c}, \boldsymbol{d} \in \mathit{Cond}_{\boldsymbol{P}}$). Monotonicity can be enforced syntactically for substitutions by never placing a variable under an odd number of negations, which is proved by structural induction as in fixed point logics or the modal μ calculus.

Definition 7 (μ-Condition). Given a finite list $\boldsymbol{P}$, if $\{\mathcal{F}_i\}_{i \in \{1,\ldots,\|P\|\}}$ are conditions with placeholders from $\boldsymbol{P}$, over the graphs of $\boldsymbol{P}$ respectively, then $\mu[\boldsymbol{P}]\boldsymbol{\mathcal{F}}$ denotes the least fixed point of the operator $\boldsymbol{c} \mapsto \boldsymbol{\mathcal{F}}[\mathtt{x}/\boldsymbol{c}]$.

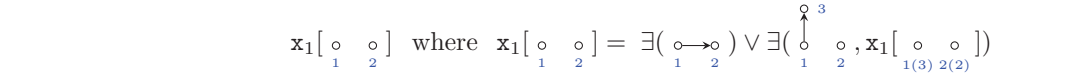
A *μ-condition* is a pair (b, l) consisting of a condition with placeholders b, and a finite list of pairs $l = (\mathtt{x}_i, \mathcal{F}_i(\boldsymbol{x}))$ of a variable $\mathtt{x}_i : P_i$ and a condition $\mathcal{F}_i(\boldsymbol{x}) : P_i$, with placeholders from $\boldsymbol{x}$, for some graph P_i, such that $\boldsymbol{\mathcal{F}}$ is monotonic.

Notation. we write the list of pairs $l = (\mathtt{x}_i, \mathcal{F}_i(\boldsymbol{x}))$ as a system of equations $\boldsymbol{x} = \boldsymbol{\mathcal{F}}(\boldsymbol{x})$. We call b the *main body* and l the *recursive specification* of (b, l), and $\boldsymbol{\mathcal{F}}(\boldsymbol{c})$ is understood as substitution of conditions $\boldsymbol{c}$ for the variables $\mathtt{x}$.

Thus each condition with placeholders typed over $\boldsymbol{P}$ defines a unary operator on $Cond_{\boldsymbol{P}}$.

Remark 3 (First Example: μ-Conditions are More General than Nested)**.**

1. μ-conditions generalise nested conditions, consequently all examples for nested conditions are examples for μ-conditions (with no variables or equations).
2. μ-conditions are strictly more general than nested conditions: the following expresses the existence of a path of unknown length between two given nodes.



It is read as follows: the word "where" separates the main body from the equations. Here, $\mathtt{x}_1$ is the only variable, and its type graph is indicated in square brackets. The second existential quantifier uses a morphism to unselect node 1 and the sole edge: its source is the type graph of $\mathtt{x}_1$, which is indeed syntactically required for using the variable in that place. The unselection morphism ι is implicit in the notation, and is only expressed by adding small blue numbers in parentheses to the node numbers in its source graph to specify the mapping. This compact notation for ι is why the second existential quantifier in the example has only two fields. To ease reading and writing, we adopt the convention to always use precisely the same layout for the type graph of a given variable.

The following statement is not needed in the proofs that will follow, but it helps motivate the use of the "unselection" morphisms. We therefore view it as justified to leave the proof as an exercise:

Remark 4 (Why ι)**.** A μ-condition where ι is the identity in all subconditions of the main body and of the components $\mathcal{F}_i(\boldsymbol{x})$ is equivalent to a nested condition.

The following fact is well-known:

Remark 5. The least fixed point of $\boldsymbol{\mathcal{F}}$ is equivalent to $\bigvee_{n\in N} \boldsymbol{\mathcal{F}}^n(\boldsymbol{false})$.

Definition 8 (Satisfaction). The μ-condition $b \mid \boldsymbol{x} = \boldsymbol{\mathcal{F}}(\boldsymbol{x})$ with $\boldsymbol{x} : \boldsymbol{P}$ is satisfied by a morphism f iff $f \models b[\boldsymbol{x}/\mu[\mathcal{P}]\mathcal{F}]$.

Remark 6 (No Infinite Nesting)**.** By the characterisation of the least fixed point as an infinite disjunction, every μ-condition is equivalent to an infinite nested condition. Infinitely deep nesting does not arise, because the characterisation in Remark 5 yields a countable disjunction of *finitely* deeply nested conditions.

A morphism satisfies a given μ-condition iff it satisfies the finite nested condition obtained by unrolling the recursive specification up to some finite depth and substituting the resulting nested conditions into the main body:

Proposition 1 (Satisfaction at Finite Recursion Depth). $f \models b \mid \boldsymbol{x} = \boldsymbol{\mathcal{F}}(\boldsymbol{x})$ *iff* $\exists n \in \mathbb{N},\ f \models b[\boldsymbol{x}/\boldsymbol{\mathcal{F}}^n(\boldsymbol{false})]$.

Theorem 1 (Deciding Satisfaction of μ-conditions). Given a morphism $f : P \hookrightarrow G$ and a μ-condition c, it is decidable whether $f \models c$.

3.2 Weakest Liberal Preconditions of μ-conditions

In this subsection, we present a construction to compute the weakest liberal precondition of any given μ-condition with respect to any graph program P that does *not* use iteration ("liberal" means that termination of P is not implied, and is redundant in the absence of iteration, as only iteration causes non-termination).

Definition 9 (Weakest Liberal Precondition). The weakest liberal precondition (wlp) of c with respect to the program P, $\mathrm{wlp}(P, c)$, is the least condition with respect to implication such that $f' \models c \Rightarrow f \models \mathrm{wlp}(P, c)$ if $(f, f', p) \in [\![P]\!]$ for some partial morphism p.

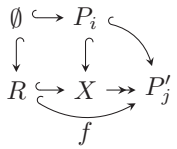
We will show that under this assumption there is a μ-condition that expresses precisely the weakest liberal precondition of a given μ-condition with respect to a rule, and it can be computed. The result is similar to the situation for nested conditions. To derive it, we use the *shift* transformation $A_m(c)$ from [12] whose fundamental property is to transform any nested condition c into another nested condition such that $m'' \models A_m(c)$ iff $m'' \circ m \models c$ for all composable pairs m'', m of monomorphisms (Lemma 5.4 from [12]). Since this and similar constructions play an important role in this section, we recall here the case $c = \exists(a, c')$: if (m', a') is the pushout of (m, a), let Epi be the set of all epimorphisms e with domain $\mathrm{cod}(m')$ that compose to monomorphisms $b := e \circ a'$ and $r := e \circ m'$. Then $A_m(\exists(a, c')) = \bigvee_{e \in Epi} \exists(b, A_r(c'))$.

With help of the unselection ι in $\exists(a, \iota, c)$, it is at first glance trivial to exhibit a weakest liberal precondition with respect to $Uns(y)$. However, to handle the addition and deletion steps, a construction becomes necessary that makes the affected subgraph explicit again. This information is crucial to obtain the weakest liberal precondition with respect to $Add(r)$ and $Del(l)$ and must not be forgotten at any nesting level in order to obtain the correct result. To that aim, we define a *partial shift* construction which makes sure that the type graph of the main body is never unselected in the μ-condition but is instead mapped in a consistent way as a subgraph of the type graph of each variable. The following serves to obtain the new type graphs containing the type of the main body:

Construction 1 (New type graphs for partial shift)**.**

We assume that an arbitrary total order on all graph morphisms is fixed. If $c = b \mid \mu[\boldsymbol{K}]\boldsymbol{F}$ is a μ-condition, then for a variable $\mathtt{x}_i$ of $\boldsymbol{K}$, $\mathcal{X}_{R,c}(\mathtt{x}_i)$ is defined as the sequence of morphisms $\boldsymbol{f}$ obtained as below, in ascending order.

The morphisms f are obtained from $\boldsymbol{P'}$ by collecting all epimorphisms that compose to monomorphisms with the pushout morphisms in the diagram:



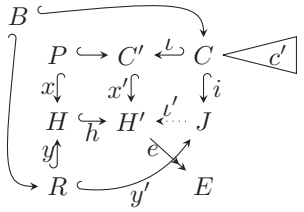
Construction 2 (Partial shift)**.**

Given monomorphisms $x : G \hookrightarrow H$ and $y : R \hookrightarrow H$,

$\mathcal{P}_{x,y}(b \mid \mu[\boldsymbol{K}]\mathcal{F}) := \mathcal{P}_{x,y}(b) \mid \mu[\boldsymbol{K'}]\mathcal{F}'$, where the new list of variables $\boldsymbol{K'}$ and their respective types $\boldsymbol{P'}$ are obtained by concatenating all $\mathcal{X}_{R,c}(\mathtt{x_i})$ of the variables of $\boldsymbol{K}$ in order. where the new variables and equations are obtained by applying $\mathcal{P}_{f,y}$ to the variables of the left hand sides with all possible morphisms f from R, as below, and accordingly to the right hand sides.

$\mathcal{P}_{x,y}(\mathtt{x}_i) = \mathtt{x}_i^y$ if $\mathtt{x}_i : G$, where $\mathtt{x}_i^y : H$ is a new variable, $H = \mathrm{cod}(y)$.

$\mathcal{P}_{x,y}(\exists(P \overset{a}{\hookrightarrow} C' \overset{\iota}{\hookleftarrow} C, c'))$ is constructed as follows: Let Epi be the set of all epimorphisms e with domain H' that compose to monomorphisms $r = e \circ x'$ and $b = e \circ h$ with the pushout morphisms. $\mathcal{P}_{x,y}(\exists(P \hookrightarrow C' \hookleftarrow C, c') = \bigvee_{Epi} \exists(H \hookrightarrow E \hookleftarrow J, \mathcal{P}_{i,y'}(c'))$: for each member of the disjunction, form the pullback of $r \circ \iota$ and $b \circ y$, then pushout the obtained morphisms to (y', i) as in the diagram:



Boolean combinations of conditions are transformed to the corresponding combinations of the transformed members.

Remark 7 (Ambiguous Variable Contexts)**.** Note that in a μ-condition it is not necessarily true that in all contexts where $\mathtt{x_i}$ is used, it appears with the same morphism $R \hookrightarrow P_i$ (where R is the type of b). It is however possible to equivalently transform every μ-condition into a "normal form" that has that property. Applying $\mathcal{P}_{id_R,id_R}$ will by construction result in a μ-condition with unambiguous inclusions $R \hookrightarrow P_i$ for all variables (namely the morphisms from the sequences $\mathcal{X}_{R,c}$), and this property is also preserved by the constructions introduced later in this section. Unreachable variables created by $\mathcal{X}$ and $\mathcal{P}$ can be pruned to obtain an equivalent, but sometimes smaller μ-condition.

Equivalence of conditions with placeholders (unlike μ-conditions) is only defined for conditions using the same sets of variables, as equivalence in the sense of nested conditions for each valuation. We extend A to conditions with placeholders by defining $A_m(\mathtt{x})$ to be $\exists(id_{\mathrm{cod}(m)}, m, \mathtt{x})$ if $\mathtt{x} : P$.

One can show that $\mathcal{P}_{x,y}$ is equivalent to A_x. The reason for introducing $\mathcal{P}_{x,y}$ is that it allows precise control over the types of the variables in the transformed condition, which should include the type of the main body. Intuitively, as this corresponds to the currently selected subgraph of a graph program, additions and deletions are applied to that subgraph and one must make sure that the changes apply to the whole μ-condition to obtain the correct result.

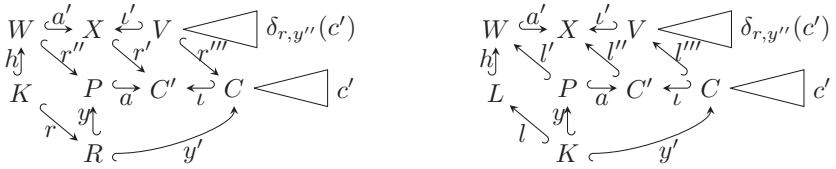
Lemma 1. The conditions $\mathcal{P}_{x,y}(c)$ and $A_x(c)$ are equivalent.

We introduce the transformations $\delta'_m(c)$, $\alpha'_m(c)$ (based on auxiliary transformations $\delta_{m,y}(c)$ and $\alpha_{m,y}(c)$, repsectively), which are used in the computation of the weakest liberal precondition (with respect to addition and deletion, respectively[4]), of a μ-condition that has already undergone partial shift:

Definition 10 (Transformations δ' and α').

Let $c : G$ be a condition with placeholders. If $r : K \hookrightarrow R$ and $y : R \hookrightarrow G$ (resp. $l : K \hookrightarrow L$ and $y : K \hookrightarrow G$) are monomorphisms, then $\delta_{r,y}(c)$ ($\alpha_{l,y}(c)$) is defined as follows: $\delta_{r,y}(\neg c) = \neg\delta_{r,y}(c)$ and $\delta_{r,y}(\bigvee_{j\in J} c_j) = \bigvee_{j\in J} \delta_{r,y}(c_j)$ (respectively: $\alpha_{l,y}(\neg c) = \neg\alpha_{l,y}(c)$ and $\alpha_{l,y}(\bigvee_{j\in J} c_j) = \bigvee_{j\in J} \alpha_{l.y}(c_j)$).

For $c = \exists(a, \iota, c')$, the following constructions are used:



Case of $\delta_{m,y}(c')$: If the pushout complement of r and $a \circ y$ does not exist, then $\delta_{m,y}(c) = false$. Otherwise, obtain it as x' and r' and pullback (a, r') to (a', r'') with source W; this yields a morphism h from K to W to make the diagram commute and the special PO-PB lemma [3] applicable. Pullback (ι, r') to (ι', r'''), $(x' = a' \circ h)$ and let $\delta_{m,y}(c) = \exists(a', \iota', \delta_{m,y''}(c'))$ (the pullback property yields existence and uniqueness of y'' between K and V to make it commute).

Case of $\alpha_{l,y}(c)$: pushout (y, l) to (l', h); pushout (l', a) to (l'', a'); pullback $(a' \circ h, l'' \circ \iota)$ and pushout to (y'', l''') over the pullback (not drawn in the figure except for the morphism l''''). The commuting morphism from the pushout object V to X fills in to yield $\exists(a', \iota', \alpha_{l,y}(c'))$. The commuting morphism from L to V is y''.

For variables, $\delta_{m,y}(\mathtt{x_i}) = \mathtt{x'_i}$ is a new variable of type K, likewise $\alpha_m(\mathtt{x_i})$ has type L (see Rem. 7). Finally, $\delta'_m(c) = \delta_{m,id}(\mathcal{P}_{id,id}(c))$ and $\alpha'_m(c) = \alpha_{m,id}(\mathcal{P}_{id,id}(c))$.

In contrast to $\mathcal{P}$, the transformations α' and δ' leave the number of variables unchanged. Only the types of the variables are modified. We recall that for any $l : K \hookrightarrow L$, there is a condition $\Delta(l)$ that expresses the possibility of effecting $Del(l)$, i.e. $\Delta(l)$ is satisfied exactly by the first components of tuples in $[\![Del(l)]\!]$. We describe $\Delta(l)$ only informally: $f \models \Delta(l)$ states the non-existence of edges that are in $im(f)$ but incident to a node in $im(f) - im(f \circ l)$.

Theorem 2 (Weakest Liberal Precondition for μ-conditions). For each rule ϱ, there is a transformation Wlp_ϱ that transforms μ-conditions to μ-conditions and assigns to each condition c such that $m' \models c$ another condition $\mathrm{Wlp}_\varrho(c)$

[4] The letters were chosen so as to indicate the effect of the transformation: to compute the weakest precondition with respect to *addition*, δ' needs to *delete* portions of the morphisms in the condition, and vice versa.

with the property that $m \models \mathrm{Wlp}_\varrho(c)$ whenever $(m', m, p) \in [\![\varrho]\!]$ and $\mathrm{Wlp}_\varrho(c)$ is the least condition with respect to implication having this property.

3.3 A Weakest Liberal Precondition Example

In this subsection, we construct a weakest liberal precondition of a μ-condition step by step. Figure 2 shows a single-rule graph program which matches a node with exactly one incoming and one outgoing edge and replaces this by a single edge. The effect of the rule is to contract paths, and it can be applied as long as no other edges are attached to the middle node.

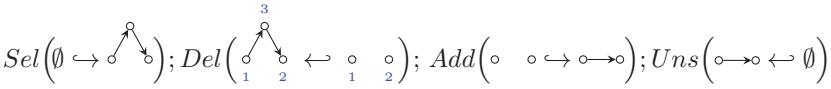


Fig. 2. A path-contracting rule $\varrho_{\mathrm{contract}} = Sel_{\mathrm{c}}; Del_{\mathrm{c}}; Add_{\mathrm{c}}; Uns_{\mathrm{c}}$.

Figure 3 shows a μ-condition whose weakest liberal precondition we wish to compute. It is a typical example of a μ-condition, which evaluates to *true* on those graphs that contain some node which has a path to every other node.

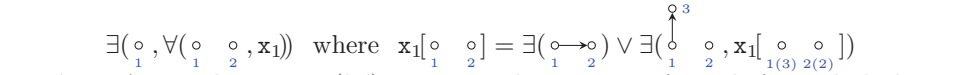


Fig. 3. A μ-condition $c_3 = (b, l)$ expressing the existence of a node from which there exists a path to every other node.

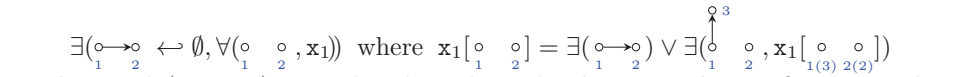


Fig. 4. $\mathrm{Wlp}(Uns_{\mathrm{c}}, c_3)$. Note that the nodes under the universal quantifier are not the same as those of the outer existential quantifier, as these have been unselected: the type of the subcondition $\forall(...)$ is $\emptyset$.

In Figures 5 and 6, a partial shift has been applied to the condition of Figure 4 ($\mathrm{Wlp}(Uns_{\mathrm{c}}, c_3)$), and the modifications the condition undergoes in the computation of the weakest precondition with respect to Add_{c} and Del_{c} are highlighted in various colours (see Figure 7 for a legend). Construction 1 has yielded a new list of variables[5], $\mathtt{x}_1, ..., \mathtt{x}_7$, the corresponding equations are shown in 6. Note that the representation is somewhat further abbreviated: type graphs of variables are suppressed in the notation in subconditions $\exists(a, \iota, \mathtt{x}_i)$, when the mapping ι from the type graph to the target of a is the identity. No other simplifications were applied. We have highlighted in yellow and red the type of the main body of $\mathrm{Wlp}(Uns_{\mathrm{c}}, c_3)$ throughout Figure 5; edges that are highlighted in red are deleted to compute $\mathrm{Wlp}(Add_{\mathrm{c}}, \mathrm{Wlp}(Uns_{\mathrm{c}}, c_3))$ as per Def. 10; the edges and nodes highlighted in red are not present initially, but added to compute $\mathrm{Wlp}(Del_{\mathrm{c}}, \mathrm{Wlp}(Add_{\mathrm{c}}, \mathrm{Wlp}(Uns_{\mathrm{c}}, c_3)))$ as per Def. 10, which is obtained by conjoining $\Delta(l)$ to the main body (which we have not represented, as it is easy to compute and would only encumber the illustration). In the end, a universal

[5] Although the original μ-condition needed only one variable, the partial shift yields a μ-condition with multiple variables in general.

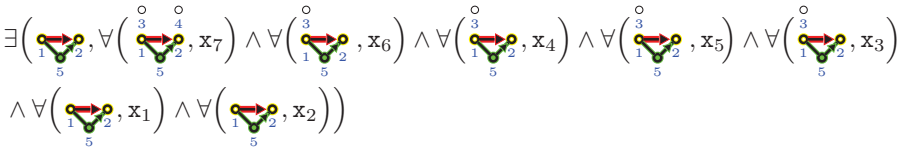


Fig. 5. Construction of $\mathrm{Wlp}(Del_c; Add_c; Uns_c, \varrho_c)$: main body (variables: see below).

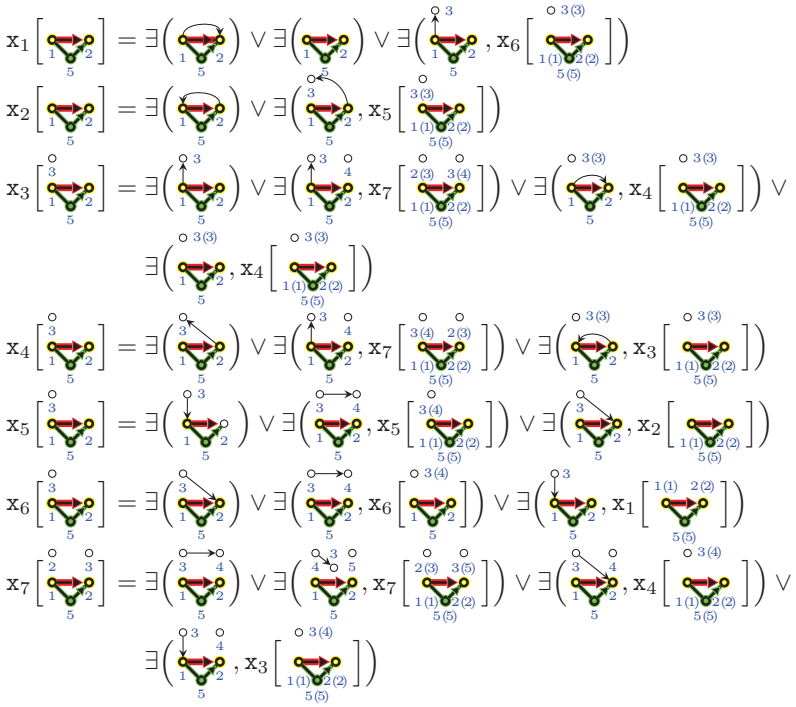


Fig. 6. Construction of $\mathrm{Wlp}(Del_c; Add_c; Uns_c, \varrho_c)$: equations for the variables.

node/edge decoration	meaning
	items (nodes and edges) selected for $Wlp(Uns(y), c)$
	items to be deleted to obtain $Wlp(Add(r), c)$
	items to be added to obtain $Wlp(Del(l), c)$

Fig. 7. Legend for Figure 5.

quantification with morphism $\emptyset \hookrightarrow L$ completes the weakest precondition with respect to the rule, as in the construction for nested conditions [12].

The outer existential quantifier in Fig. 5, where the unselection morphism is not shown in the abbreviated representation, is really as in Fig. 8:

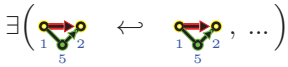


Fig. 8. Outer nesting level of the conditions in Fig. 5

When following the construction through the nesting levels, please keep in mind that one may sometimes choose among isomorphic pushout objects and the numbers of new nodes are arbitrary, but the nodes 1, 2 and (as created by the transformation α') 5 are never "unselected" and therefore present in every type graph occurring in the weakest preconditions, similarly for the edges (not numbered because their mapping is unambiguous in the example).

4 Correctness Relative to μ-conditions

In the previous section, we have shown how the weakest liberal precondition construction for nested conditions carries over to μ-conditions. The next task is to develop methods for deducing correctness relative to μ-conditions and extend the proof calculus, for which we offer a partial solution in this section.

The soundness of Pennemann's calculus $\mathcal{K}$ has been established in the publications introducing them, and recently a tableaux based completeness proof of $\mathcal{K}$ was published [11]. The proof rules of $\mathcal{K}$ are easily seen to be sound for μ-conditions as well, however the recursive definitions requires an extension.

For our calculus $\mathcal{K}_\mu$, we adopt the resolution-style rules of $\mathcal{K}$ and add an induction principle to deal with certain situations involving fixed points. This proved to be sufficient to handle all the situations encountered in the examples.

We employ a sequent notation: the inference rules manipulate objects $Ctx : \Gamma \vdash \Delta$, with the intended meaning that the disjunction of Δ can be deduced from the conjunction of Γ in the context Ctx. The context Ctx is a pair of a left hand side of a sequent and an operator on μ-conditions. Γ and Δ are sets of expressions, which differ from conditions in that identifiers can be used for the main bodies of μ-conditions and both these and variables can be annotated with their recursion depth, an implicitly universally quantified natural number. $\mathtt{x}_i^{(n)}$ then stands for $\mathcal{F}_i^{(n)}(\mathit{false})$ and an auxiliary rule permits to unroll it to the i-th component of $\boldsymbol{\mathcal{F}}$ applied to $\mathtt{x}^{(n-1)}$.

The induction rule announced above is (where $\mathcal{H}_{i,j}$ for each $i \in \{1, ..., \|I\|\}$, $j \in \{1, ..., \|J\|\}$ is any condition with placeholders):

$$\frac{\overbrace{\begin{array}{l}\mathtt{x}_i^{(m)} \wedge \neg \mathtt{y}_j^{(n)} \vdash \\ \mathcal{H}_{i,j}(\mathtt{x}_1^{(m-1)} \wedge \neg \mathtt{y}_1^{(n)}, ..., \mathtt{x}_{\|I\|}^{(m-1)} \wedge \neg \mathtt{y}_{\|J\|}^{(n)})\end{array}}^{\forall i,j} \quad \begin{array}{l}\vdash \\ \forall i, j\ \mathcal{H}_{i,j}(\boldsymbol{false}) = \boldsymbol{false}\end{array}}{\bigvee \mathtt{x}_i \wedge \neg \mathtt{y}_i \vdash \mathit{false}}$$

(IndMuEmpty)

Theorem 3. $\mathcal{K}_\mu := \mathcal{K} \cup \{\text{IndMuEmpty}\}$ is sound.

There are a number of details hidden in the discussion above: Boolean operations must be lifted to μ-conditions, which entails variable renaming and union of the

systems of equations; rules for exploiting logical equivalences between different Boolean combinations are necessary to equivalently transform conditions into a form suitable for the application of the rules of $\mathcal{K}$; in [12], each Boolean combination appearing inside a nested condition is put into conjunctive normal form prior to the application of rules. Proof trees in the sequent-style calculus $\mathcal{K}_\mu$ start with instances of the *axiom* ($A \vdash A$ is derivable by a rule with no antecedents), and make use of all the classical sequent rules [5] not involving quantifiers.

Our handling of nested contexts relies on substitutions: a context is a pair of a left hand side of a sequent, and a graph condition with a special variable. The rule for manipulating the context is usable both ways:

$$\frac{\vdash Ctx(x)}{Ctx \vdash x} \qquad \text{(Ctx)}$$

5 Related Work

Recently, Poskitt and Plump [16] have presented a weakest precondition calculus for a different extension of nested conditions (monadic second-order conditions) and demonstrated how to use it in a Hoare logic. The method is arguably closer to reasoning directly in a logic and less graph condition like, but seems successful at solving some of the same problems in a different way. HR* conditions [17] are another approach towards the same goal; they have already been mentioned in the main text and recently there has been an effort at extending the weakest precondition calculus to a subclass including path expressions. Verification of graph transformation system has also been performed within general-purpose theorem proving environments by Strecker et al. [19, 13], with positive path conditions. Verification of graph transformation systems via model checking of abstractions, as opposed to the prover-based approach pursued here, can be found in Gadducci et al., Baldan et al., König et al., Rensink et al. [4, 1, 10, 18].

A summary overview of graph conditions for non-local properties is attempted below (a proof calculus is presented in [16] but completeness of a proof calculus has only recently been obtained by Lambers and Orejas [11] for nested conditions and remains to be researched for the other approaches). Note that while HR* conditions are known to properly contain the monadic second-order definable properties [17] and nested conditions are a special case of each of the other three, we have not yet been able to separate μ-conditions from MSO or HR*:

reference	[12]	(here)	[17]	[16]
conditions	Nested	μ-	HR*	MSO-
wlp	yes	yes	incomplete[6]	yes
complete proof calculus	yes	future work		
theorem prover	yes	future work		

[6] Radke, personal communication.

6 Conclusion and Outlook

We have introduced μ-conditions and achieved several results, mainly a weakest liberal precondition transformation (Theorem 2), soundness of a proof calculus (Theorem 3), and discussed correctness relative to μ-conditions, which appears to be a fruitful ground for further investigations.

In analogy to the equivalence between first-order predicate graph logic and nested graph conditions, we are investigating whether μ-conditions have the same expressivity as fixed point extensions to classical first-order logic for finite graphs.

Also, the expressivity of HR* conditions [17] or even MSO likely surpasses that of μ-conditions, but the precise relationship remains to be examined. As the examples show, the weakest precondition calculus (which is still a research question for HR* conditions [17] but readily available by logical means in the MSO-conditions formalism [16]) produces quite unwieldy expressions due to partial shift. The blowup is exponential in the size of the interface graphs used in the rule, and seems unavoidable because of the need to use a fixed set of type graphs for the finitely many variables (and a blowup is also inherited from the weakest precondition calculus of [12]). Rule IndMuEmpty also contributes because it involves a Cartesian product between variable sets. We have devised heuristics to simplify the expressions, but even if many of the cases can be resolved automatically, this issue still raises concerns as to the practical applicability.

Future work will also include tool support with special attention to semi-automated reasoning, based on the reasoning engine Enforce implemented in [12]. To extend the weakest liberal precondition construction to programs with iteration, one would have to provide, or have the prover attempt to determine, an invariant, as in the original work of Pennemann; to obtain termination, one could proceed as in [14] and prove a termination variant. We plan a further generalisation to correctness under adverse conditions, i.e. systems subject to environmental interference, also modelled as a graph program. Furthermore, it appears that μ-conditions might readily generalise to temporal properties, even with the option to nest temporal operators inside quantifiers, which would allow properties such as the preservation of a specific node to be expressed (but require further proof rules). This could be achieved by introducing a temporal *next* operator parameterised on atomic subprograms (the basic steps of Def. 4) and since in the semantics of these program steps the relationship between the interfaces is deterministic, this would again confer an unambiguous *type* to such an expression and make it suitable for use as a subcondition. Whether this offers any new insights remains to be seen. Eventually, we would also like to deal with algebraic operations on attributes and extend our work to a practical verification method that separates the graph specific concerns from other aspects and allows proofs of properties that depend on both, for example involving data structures whose elements should remain ordered. Finally, the limitations imposed by undecidability prompt the search for of restricted decidable classes.

Acknowledgements

We thank Annegret Habel, many other members of SCARE and the anonymous reviewers for constructive criticism of the approach and the paper.

References

1. Baldan, P., König, B., König, B.: A logic for analyzing abstractions of graph transformation systems. In: Static Analysis, pp. 255–272. Springer (2003)
2. Dijkstra, E.W.: A discipline of programming. Prentice Hall (1976)
3. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science, Springer (2006)
4. Gadducci, F., Heckel, R., Koch, M.: A fully abstract model for graph-interpreted temporal logic. In: TAGT'98. LNCS, vol. 1764, pp. 310–322 (1998)
5. Gentzen, G.: Untersuchungen über das logische Schließen. I. Mathematische Zeitschrift 39(1), 176–210 (1935)
6. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. Math. Struct. in Comp. Sci. 19(2), 245–296 (2009)
7. Habel, A., Pennemann, K.H., Rensink, A.: Weakest preconditions for high-level programs. In: ICGT 2006. LNCS, vol. 4178, pp. 445–460 (2006)
8. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 26(1), 53–56 (1983)
9. Kreutzer, S.: Pure and Applied Fixed-Point Logics. Ph.D. thesis, Dissertation thesis, RWTH Aachen (2002)
10. König, B., Kozioura, V.: Counterexample-guided abstraction refinement for the analysis of graph transformation systems. LNCS, vol. 3920, pp. 197–211 (2006)
11. Lambers, L., Orejas, F.: Tableau-based reasoning for graph properties. In: Graph Transformation, LNCS, vol. 8571, pp. 17–32 (2014)
12. Pennemann, K.H.: Development of Correct Graph Transformation Systems. Ph.D. thesis, Universität Oldenburg (2009)
13. Percebois, C., Strecker, M., Tran, H.N.: Rule-level verification of graph transformations for invariants based on edges' transitive closure. In: SEFM 2013. LNCS, vol. 8137, pp. 106–121 (2013)
14. Poskitt, C.M.: Verification of Graph Programs. Ph.D. thesis, University of York (2013)
15. Poskitt, C.M., Plump, D.: Verifying total correctness of graph programs. Electronic Communications of the EASST 61 (2013)
16. Poskitt, C.M., Plump, D.: Verifying monadic second-order properties of graph programs. In: Graph Transformation, LNCS, vol. 8571, pp. 33–48 (2014)
17. Radke, H.: HR* graph conditions between counting monadic second-order and second-order graph formulas. Electronic Communications of the EASST 61 (2013)
18. Rensink, A., Distefano, D.: Abstract graph transformation. ENTCS 157(1), 39–59 (2006)
19. Strecker, M.: Modeling and verifying graph transformations in proof assistants. ENTCS 203(1), 135–148 (2008)
20. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Math. 5(2), 285–309 (1955)