Automated Checking and Completion of Backward Confluence for Hyperedge Replacement Grammars

Ira Fesefeldt[1][0000−0001−7837−2611] ✉, Christoph Matheja[2][0000−0001−9151−0441], Thomas Noll[1][0000−0002−1865−1798], and Johannes Schulte

[1] Software Modeling and Verification Group, RWTH Aachen University, Germany
{fesefeldt,noll}@cs.rwth-aachen.de
[2] Programming Methodology Group, ETH Zürich, Switzerland
cmatheja@inf.ethz.ch

Abstract. We present a tool that checks for a given context-free graph grammar whether the corresponding graph reduction system in which all rules are applied backward, is *confluent*—a question that arises when using graph grammars to guide state space abstractions for analyzing heap-manipulating programs; confluence of the graph reduction system then guarantees the abstraction's uniqueness. If a graph reduction system is *not* confluent, our tool provides symbolic representations of counterexamples to confluence, i.e., non-joinable critical pairs, for manual inspection. Furthermore, it features a heuristics-based completion procedure that attempts to turn a graph reduction system into a confluent one without invalidating the properties mandated by the abstraction framework. We evaluate our implementation on various graph grammars for verifying data structure traversal algorithms from the literature.

Keywords: Graph grammars · Confluence · Critical pairs · Completion.

1 Introduction

Confluence is a central property of many rewriting formalisms, including term rewriting and graph transformation systems: Confluent systems require no backtracking since all terminating sequences of rule applications produce the same result. In this paper, we present a tool that checks confluence for certain *graph reduction systems*—more precisely: hyperedge replacement grammars (HRG) [9] in which all rules are reversed—based on the algorithm in [14].

Our work is motivated by the usage of HRGs as an abstraction mechanism for verifying pointer programs. This approach is at the core of Attestor[3]—a graph-based model-checking tool for analyzing Java programs operating on dynamic data structures [2,12]. To cope with large or even unbounded state spaces arising in this context, Attestor performs a symbolic shape analysis based on a user-supplied HRG that characterizes the data structures handled by

[3] https://github.com/moves-rwth/attestor

the program. Here, a suitable HRG generates graphs modeling concrete heaps where hyperedges labeled with nonterminal symbols act as placeholders for the (partial) data structures under consideration, e.g., doubly-linked lists or binary trees with a fixed root. Abstracting of a concrete heap then corresponds to applying HRG rules backward until a normal form is reached. While termination of this procedure is guaranteed for the HRGs admitted in our setting, *confluence*, i.e., uniqueness of normal forms, is not. Confluence is vital for the performance of verification tools, such as Attestor, because—rather than abstracting a heap in all possible ways—it suffices to apply abstraction rules exhaustively and in arbitrary order. Furthermore, confluence can be exploited to decide whether an abstract state is subsumed by an already computed one—a particular instance of the graph language inclusion problem that is crucial for ensuring termination of the overall analysis in the presence of loops or recursive procedures (cf. [21]).

Apart from checking whether the graph reduction system induced by an HRG is confluent, our tool supports a heuristics-based completion procedure to transform it into a confluent one. In particular, the heuristics can be chosen such that properties of the HRG required by Attestor, e.g., those ensuring termination of the abstraction, are preserved during completion. We evaluate our implementation on various heap abstractions that have been proposed in the literature (as HRGs or equivalent inductive predicates in separation logic).

Related tools. While algorithms for deciding confluence have been extensively studied in the context of graph transformations (cf. [6,11,14,20,23,24]), there are, to the best of our knowledge, only few tools that support computing critical pairs—a key component for confluence checking.

However, we are not aware that any of the tools below support proving backward confluence for HRGs, which additionally requires checking whether the computed critical pairs are joinable. Moreover, they do not support completion.

AGG[4] is a development environment for attributed graph transformation systems supporting an algebraic approach to graph transformation [25,27]. For analyzing critical pairs, it implements the algorithm developed in [11].

VeriGraph[7] is a tool for simulation and analysis of transformation systems given by graph grammars, which appears not to be developed further anymore. It implements the critical-pair analysis described in [19]. A performance comparison with AGG is given in [4], analyzing both critical pairs and sequences to capture conflicts and dependencies between rules. The evaluation shows that Verigraph outperforms AGG in realistic test cases, which indicates that AGG is more sensitive to the size of the graphs contained in rewriting rules.

Henshin[1] is a model transformation environment that is based on the Eclipse Modeling Framework. It first integrated a critical-pair analysis as presented in [5], which has later been superseded by a more efficient and flexible conflict and dependency analysis [18].

[4] `https://www.user.tu-berlin.de/o.runge/agg/`

SyGraV[8] is a graph analysis tool that supports checking local confluence of attributed graph transformation systems. It was the first to fill the gap between theoretical results and practical usability of symbolic graph analysis.

2 The Tool

We implemented a confluence checking and a completion component within the software model checker Attestor; our confluence checker can also be used as a standalone tool. The tool, its source code, and our benchmarks are available online.[5] In this article, we do not distinguish between Attestor and its confluence checker. This section briefly outlines our tool's *input*, its main steps for *proving confluence*, the *feedback* it provides, and the extent to which it supports automatic *completion* of graph grammars. A detailed discussion of the underlying algorithm (which is based on [14]), its implementation, and the heuristics applied for guiding completion is found in [26].

2.1 Input

Our confluence checker targets a subset of HRGs suitable for modeling dynamic data structures (cf. [12,2] for details). While HRGs are context-free graph grammars—and thus always confluent—Attestor checks whether the corresponding *graph reduction system* (GRS) [13] in which all rules are applied in reverse direction, is confluent as well. In our setting, graph rewriting through reverse rule applications always terminates because we require all HRG rules to be increasing, i.e., every hyperedge connected to $n \geq 0$ nodes is mapped to a hypergraph consisting of at least $n + 1$ nodes and edges. HRGs are specified as a set of (forward) rules in a JSON-style format. Both nodes and hyperedges are equipped with attributes indicating the type of nodes and edges; these types can be consulted to differentiate elements when checking for graph isomorphism.

2.2 Proving Confluence

Attestor implements the confluence checking algorithm in [14] for the set of GRSs from above. That is, it systematically computes all *critical pairs*—overlappings of two hypergraphs appearing on the left-hand side of graph transformation rules—and, for each critical pair, checks whether it is *strongly joinable*, i.e., exhaustive rewriting after applying either of the two possible rules leads to isomorphic normal forms. Here, "strongly" refers to an additional requirement while searching for graph isomorphisms: we distinguish between nodes in the original overlapping that are not deleted by rule applications. The above condition is necessary because a GSR is confluent iff all critical pairs are strongly joinable. Attestor reports if a critical pair is joinable but not strongly joinable since this case seems to be a frequent error when manually designing supposedly confluent grammars for heap abstraction (see evaluation).

[5] `https://github.com/moves-rwth/attestor-confluence`

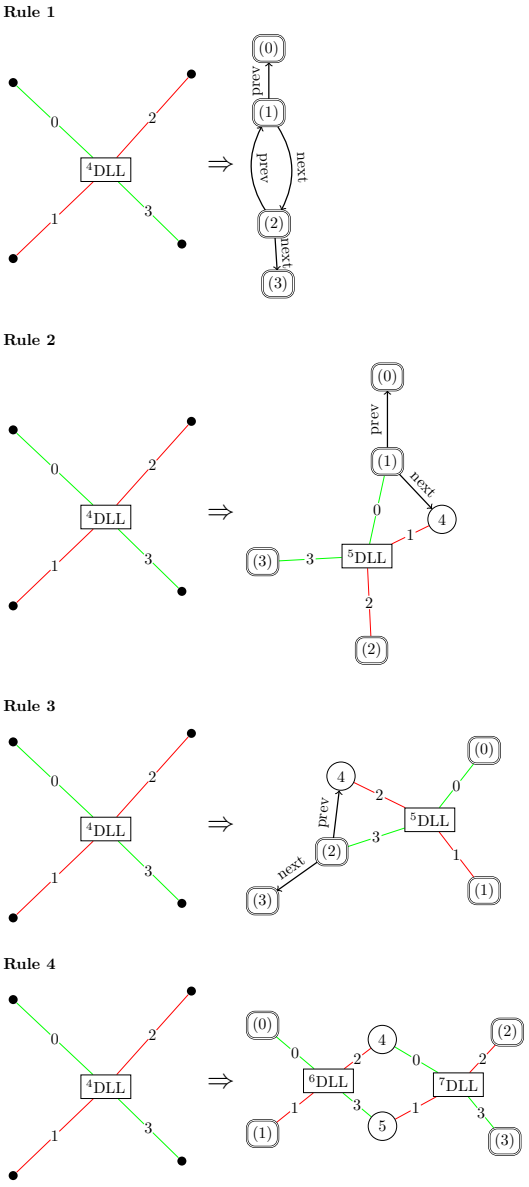

Fig. 1. Rules of an HRG modelling doubly-linked list segments; the graphical representation was automatically generated by Attestor. In our graphical notation, circles indicate nodes; double circles capture the sequence of external nodes (ordered by their label). Hyperedges are drawn as rectangles; the numbered connections indicate the sequence of nodes attached to a hyperedge, i.e., all hyperedges labeled with DLL are attached to four (not necessarily different) nodes. For simplicity, hyperedges connected to exactly two nodes are drawn as (labeled) arrows. In all hypergraphs, nodes and edges are additionally equipped with an integer, e.g., 4 for the single hyperedge on the LHS of rules), to identify them across rule applications.

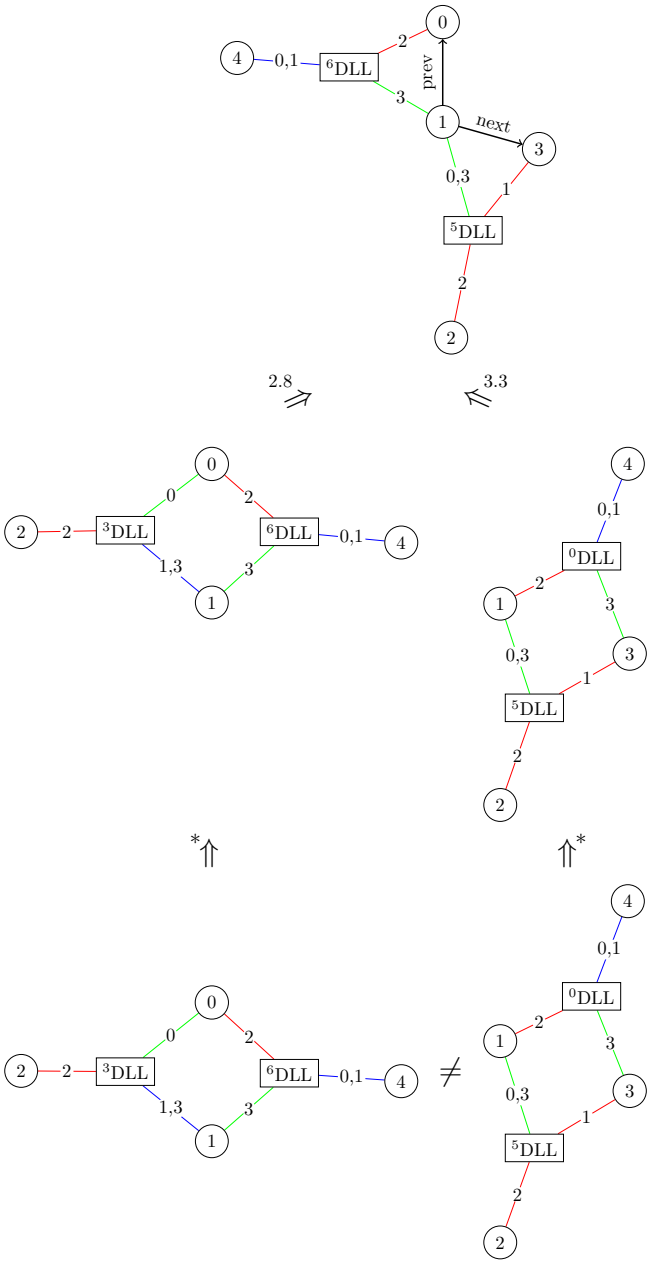

Fig. 2. Example of the feedback generated for a critical pair. It displays a non-joinable pair at the bottom, a trivial derivation to the graphs in the middle and a derivation using the second and third rule from Figure 1 to obtain the context graph of the critical pair at the top.

2.3 Graphical Feedback

Apart from answering whether a given GRS is confluent, ATTESTOR generates a report (in LaTeX, using the TikZ library) that visualizes for each critical pair: (a) the context graph, i.e., the overlapping of two left-hand sides of rules, (b) the two rules that are applied, (c) the graphs obtained after one rule application, and (d) at most two normal forms obtained after further exhaustive rule applications up to isomorphism. If a GRS is not confluent, the report lists counterexamples consisting of non-isomorphic normal forms for the same critical pair.

Figure 2 depicts an excerpt of ATTESTOR's output for a single critical pair. The underlying HRG generates non-empty doubly-linked list segments as shown in Figure 1. Each edge labeled with DLL represents a doubly-linked list segment that is connected with the predecessor of the previous node (0), the previous node (1), the next node (2) and the successor of the next node (3). The first rule generates a doubly-linked list of length 2 (the smallest this grammar can generate assuming the first and last node are both the "null" node). The second and third rule represent graphs obtained from traversing the list in forward and backward direction, respectively. The fourth rule was introduced to achieve backward confluence: intuitively, it states that two correctly connected doubly-linked lists segments again represent a doubly-linked list segment.

However, as our analysis shows, the fourth rule is not sufficient to guarantee confluence. In Figure 2, the context graph is at the top and the graphs resulting from the first two rule applications (2.8 and 3.3)[6] are directly below; the numbers assigned to each node and hyperedge serve to identify them throughout rule applications. The critical pair in question is not strongly joinable because there is no isomorphism between the two graphs at the bottom, which are obtained after exhaustive rule application. In particular, the hyperedges 6 and 0 are attached to different sequences of nodes.

2.4 Automated Completion

ATTESTOR also supports a simple completion procedure to turn. a given GRS into a confluent one. In contrast to existing completion algorithms, such as Knuth-Bendix [17], we are not interested in *any* extension that ensures confluence. Instead, reversing all GRS rules should still lead to an HRG that meets ATTESTOR's requirements for heap abstractions.

We thus opted for implementing a greedy procedure that applies various heuristics which preserve the aforementioned HRG properties. By choosing suitable heuristics, our greedy approach enables rapid prototyping of handcrafted strategies for devising appropriate heap abstractions. For example, one heuristic attempts to add rules that group connected hyperedges with identical labels into a single one of the same label.

[6] That is, rules 2 and 3 were applied. To improve performance, ATTESTOR generates specialized rules in which two or more external nodes are identical; the number after the dot indicates which case of a rule has been applied.

Table 1. The experimental results for our confluence checker; confluent grammars are marked with ✓. All runtimes are in milliseconds.

	Critical Pairs				Runtime			
Grammar	Not	Weak	Strong	Total	Node	Edge	Validity	Total
✗ `InTree`	22	1	18	41	9.6	19.6	32.8	63.5
✓ `InTreeLinked`	0	0	83	83	11.1	7.0	32.8	51.7
✗ `LinkedTree1`	5	0	10	15	6.0	9.3	4.0	19.7
✗ `LinkedTree2`	217	4	20	241	61.1	111.5	95.7	270.1
✓ `BT`	0	0	33	33	2.1	4.8	4.9	12.2
✓ `SLList`	0	0	15	15	0.3	0.4	0.8	1.6
✗ `SimpleDLL`	1	0	2	3	0.1	0.4	0.2	0.7
✗ `DLList`	63	12	137	212	89.8	22.5	78.7	192.6

3 Evaluation

We evaluated our implementation on both confluent and non-confluent graph grammars proposed for modeling dynamic data structures. For the non-confluent grammars, we also experimented with feeding our greedy completion procedure with various heuristics to turn them into confluent ones.

Setup. All experiments were performed on a Thinkpad X1 Carbon 2019 with an Intel Core i7-8565U, 1.8 GHz and 16 GB Ram, which runs Ubuntu 20.04.1.

Confluence Checking. Table 1 shows our experimental results for checking whether a given graph reduction system induced by an HRG is confluent. As noted in the previous section, Attestor checks for each critical pair whether it is *strongly-joinable*, *weakly joinable* or *not joinable* at all. To conclude that a graph grammar is confluent (and thus mark it with ✓), all critical pairs must be strongly joinable. Furthermore, we measured the time for computing *overlappings of nodes*, *overlappings of edges*, *a validity check* for possibly spurious pairs (we discard graphs that do not model heaps and thus cannot appear in our setting) as well as the total runtime. Starting the java virtual machine and parsing the grammar took 0.9s CPU time and was thus dominant for small examples.

The HRGs `InTree` [15] and `InTreeLinked` generate "in-trees", i.e., binary trees in which the direction of edges is inverted such that child nodes point to their parent. `InTreeLinked` additionally connects all leaves of the in-tree from left to right via a singly-linked list. It is noteworthy that—despite having more than twice as many rules—Attestor managed to prove confluence for `InTreeLinked` faster than determining all critical pairs that are not strongly joinable for `InTree`. One possibe explanation is that the different edge labels (for the left and right child as well as the successor in the list) used by `InTreeLinked` reduce the number of edge overlappings that need to be computed.

The HRGs `LinkedTree1` and `LinkedTree2` generate binary trees with a given root, where each node has a back pointer to its parent and all leaves are connected from left to right. While `LinkedTree2`, which is taken from [22], consists of only

Table 2. Completion results (✓ if successful, ✗ if unsuccessful) for different combinations of completion heuristics. All runtimes are in seconds.

	InTree	LinkedTree1	LinkedTree2	SimpleDLL	DLList
CA	✗, 0.003	✗, 0.003	✗, 0.337	✗, <0.001	✗, 0.127
RNN	✗, 1.538	✗, 0.048	✗, 55762.111	✗, 0.002	✗, 533.773
JGN	✗, 2.415	✗, 0.051	✗, 53327.031	✗, 0.003	✗, 540.818
SNR	✗, 1.576	✗, 0.048	✗, 55341.839	✓, 0.006	✗, 526.315
OR	✗, 1.640	✗, 0.046	✗, 55018.761	✓, 0.004	✗, 527.410
ORL	✗, 0.847	✗, 0.213	✗, 31075.274	✗, 0.004	✗, 362.976
A1	✗, 1.570	✓, 0.049	✓, 55042.314	✓, 0.004	✗, 392.728
A1L	✗, 1.175	✓, 0.203	✗, 27297.678	✗, 0.004	✗, 258.643
A2	✓, 0.333	✓, 0.024	✓, 12.405	✓, 0.004	✗, 26.421
A2L	✗, 0.838	✓, 0.038	✗, 27053.035	✗, 0.004	✗, 261.090

two rules with up to 7 nodes and 7 hyperedges (of rank at most 4), it turned out to be quite complex, leading to a large number of possible non-joinable critical pairs. `LinkedTree1` is an early attempt to turn a simplified version of `LinkedTree2` into an HRG that induces a confluent GRS.

The HRG `BT` [3] generates binary trees with a given root that has been handcrafted for verifying tree traversal algorithms. Ensuring confluence required two different nonterminal symbols and fourteen rules in total. Similarly, `SLList` is a handcrafted grammar modeling singly-linked lists.

`SimpleDLL` and `DLL` generate doubly-linked lists. The former version only admits list traversals from left to right whereas the latter version admits traversals in both directions. Both HRGs do *not* induce confluent GRSs. This surprised us as `DLL` has been successfully applied for analyzing pointer programs [2]. While confluence is not required for the soundness of such program analyses, non-confluence typically leads to performance penalties. Upon closer inspection, we discovered that the list-manipulating programs analyzed in [2] with these grammars did not lead to states containing a non-joinable critical pair.

Completion. We applied our confluence completion procedure to the non-confluent grammars shown in Table 1, i.e., `InTree`, `LinkedTree1`, `LinkedTree2`, `SimpleDLL` and `DLList`. The results of our tests are given in Table 2.

`CA` adds application conditions to rule out some non-joinable critical pairs; `RNN` introduces new nonterminals to join critical pairs. Moreover, `JGN` and `SNR` extend `RNN` by joining or using existing nonterminals. The remaining heuristics combine subsets of the above heuristics in different order (details are found in [10,26]). We require that heuristics ending with 'L' preserve local concretizability—a property ensuring that those parts of the heap that can be manipulated by one program instruction are obtainable using a single rule application (cf. [16]).

We observe that combining heuristics increases the chance to successfully complete a grammar, as is the case for `InTree`, `LinkedTree1` and `LinkedTree2`. Moreover, enforcing local concretizability can both increase runtime (see `A2` vs. `A2L`), but also affect the completion result (see `InTree`, `LinkedTree2`, and

`SimpleDLL`). For complex grammars, such `LinkedTree2`, completion is expensive, which prohibits its usage with every invocation of Attestor. In such cases, it is preferable to perform completion as a preprocessing step and store the completed grammar for re-use.

4 Conclusion

We presented a tool for checking backward confluence of (certain) attributed hyperedge replacement grammars. The tool is implemented as a component of the graph-based software model checker Attestor but can also be used stand-alone. Within Attestor, checking for backward confluence of user-supplied grammars did not lead to substantial performance penalties. However, it did improve the overall verification pipeline in at least two aspects: First, incompleteness issues, i.e., failed verification attempts of correct programs; our tool enables detecting such issues before running an expensive state space generation. Second, rather than manually inspecting thousands of graphs in a generated state space, our tool creates a counterexample to backward confluence that allows fixing the supplied grammar.

Furthermore, we implemented a heuristics-driven algorithm that attempts to turn a given HRG into a backward confluent one. Although this algorithm is expensive, it can be run independently of the actual verification pipeline; if completion succeeds, the resulting grammar can be exported for future use.

A possible future improvement of our tool's performance is to detect and omit critical pairs that are not relevant for Attestor's state space generation. Such an approach would amount to proving confluence up to garbage [6].

References

1. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced concepts and tools for in-place emf model transformations. In: MODELS 2010. LNCS, vol. 6394, pp. 121–135. Springer (2010)
2. Arndt, H., Jansen, C., Katoen, J.P., Matheja, C., Noll, T.: Let this graph be your witness! In: CAV 2018. LNCS, vol. 10982, pp. 3–11. Springer (2018)
3. Arndt, H., Jansen, C., Matheja, C., Noll, T.: Graph-based shape analysis beyond context-freeness. In: SEFM. LNCS, vol. 10886, pp. 271–286. Springer (2018)
4. Azzi, G.G., Bezerra, J.S., Ribeiro, L., Costa, A., Rodrigues, L.M., Machado, R.: The verigraph system for graph transformation. In: Graph Transformation, Specifications, and Nets, LNCS, vol. 10800, pp. 160–178. Springer (2018)
5. Born, K., Arendt, T., Heß, F., Taentzer, G.: Analyzing conflicts and dependencies of rule-based transformations in henshin. In: FASE 2015. LNCS, vol. 9033, pp. 165–168. Springer (2015)
6. Campbell, G., Plump, D.: Confluence up to garbage. In: ICGT. LNCS, vol. 12150, pp. 20–37. Springer (2020)
7. Costa, A., Bezerra, J., Azzi, G., Rodrigues, L., Becker, T.R., Herdt, R.G., Machado, R.: Verigraph: A system for specification and analysis of graph grammars. In: SBMF 2016. LNCS, vol. 10090, pp. 78–94. Springer (2016)

8. Deckwerth, F.: Static Verification Techniques for Attributed Graph Transformations. Ph.D. thesis, Technische Universität, Darmstadt (2017)
9. Drewes, F., Kreowski, H.J., Habel, A.: Hyperedge replacement graph grammars. In: Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations, pp. 95–162. World Scientific (1997)
10. Fesefeldt, I.: moves-rwth/attestor-confluence, `https://github.com/moves-rwth/attestor-confluence/blob/master/readme/HEURISTICS.md`
11. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of typed attributed graph transformation systems. In: ICGT 2002. LNCS, vol. 2505, pp. 161–176. Springer (2002)
12. Heinen, J., Jansen, C., Katoen, J.P., Noll, T.: Verifying pointer programs using graph grammars. Science of Computer Programming **97**(1), 157–162 (2015)
13. Hoffmann, B., Plump, D.: Implementing term rewriting by jungle evaluation. RAIRO Theor. Informatics Appl. **25**, 445–472 (1991)
14. Hristakiev, I., Plump, D.: Checking graph programs for confluence. In: STAF 2017. LNCS, vol. 10748, pp. 92–108. Springer (2018)
15. Jansen, C.: Static Analysis of Pointer Programs - Linking Graph Grammars and Separation Logic. Ph.D. thesis, RWTH Aachen University, Germany (2017)
16. Jansen, C., Heinen, J., Katoen, J.P., Noll, T.: A local greibach normal form for hyperedge replacement grammars. In: LATA. pp. 323–335. Springer, Berlin (2011)
17. Knuth, D., Bendix, P.: Simple word problems in universal algebra. Computational Problems in Abstract Algebra pp. 263—297 (1970)
18. Lambers, L., Born, K., Kosiol, J., Strüber, D., Taentzer, G.: Granularity of conflicts and dependencies in graph transformation systems: A two-dimensional approach. Journal of Logical and Algebraic Methods in Programming **103**, 105–129 (2019)
19. Lambers, L., Ehrig, H., Orejas, F.: Conflict detection for graph transformation with negative application conditions. In: ICGT 2006. LNCS, vol. 4178, pp. 61–76. Springer (2006)
20. Lambers, L., Ehrig, H., Orejas, F.: Efficient conflict detection in graph transformation systems by essential critical pairs. Electronic Notes in Theoretical Computer Science **211**, 17–26 (2008)
21. Matheja, C.: Automated reasoning and randomization in separation logic. Ph.D. thesis, RWTH Aachen University, Germany (2020)
22. Matheja, C., Jansen, C., Noll, T.: Tree-like grammars and separation logic. In: APLAS. LNCS, vol. 9458, pp. 90–108. Springer (2015)
23. Plump, D.: Confluence of Graph Transformation Revisited. Springer (2005)
24. Plump, D.: Checking graph-transformation systems for confluence. ECEASST **26** (2010)
25. Runge, O., Ermel, C., Taentzer, G.: Agg 2.0 – new features for specifying and analyzing algebraic graph transformations. In: AGTIVE 2011. LNCS, vol. 7233, pp. 81–88. Springer (2012)
26. Schulte, J.: Automated Detection and Completion of Confluence for Graph Grammars. Master thesis, RWTH Aachen University, Germany (2019), `http://www-i2.informatik.rwth-aachen.de/pub/index.php?type=download&pub_id=1765`
27. Taentzer, G.: Agg: A graph transformation environment for modeling and validation of software. In: AGTIVE 2003. LNCS, vol. 3062, pp. 446–453. Springer (2004)