RWTH AACHEN UNIVERSITY

The present work was submitted to the Chair for Software Modeling and Verification

MASTER OF SCIENCE THESIS

AUTOMATED DETECTION AND COMPLETION OF CONFLUENCE FOR GRAPH GRAMMARS

Johannes Schulte

1st Examiner:
Prof. Dr. Thomas Noll

2nd Examiner:
Prof. Dr. Ir. Dr. h. c. Joost-Pieter Katoen

Advisor:
Christoph Matheja

Aachen, November 4, 2019

Eidesstattliche Versicherung
Statutory Declaration in Lieu of an Oath

Schulte, Johannes

Name, Vorname/Last Name, First Name

Matrikelnummer (freiwillige Angabe)
Matriculation No. (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende ~~Arbeit/Bachelorarbeit/~~ Masterarbeit* mit dem Titel
I hereby declare in lieu of an oath that I have completed the present ~~paper/Bachelor thesis~~/Master thesis* entitled

Automated Detection and Completion of Confluence for Graph Grammars

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
independently and without illegitimate assistance from third parties (such as academic ghostwriters). I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Aachen, November 4, 2019

Ort, Datum/City, Date

Unterschrift/Signature

*Nichtzutreffendes bitte streichen

*Please delete as appropriate

Belehrung:
Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.
Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.
§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.
Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1) If a person commits one of the offences listed in sections 154 through 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:
I have read and understood the above official notification:

Aachen, November 4, 2019

Ort, Datum/City, Date

Unterschrift/Signature

Abstract

In this thesis, we present an algorithm that determines if a hyperedge replacement grammar (HRG) is backward confluent. A backward confluent HRG has the property that all exhaustive backward rule applications always lead to the same graph. This means backtracking is not necessary when exhaustively applying rules backward, as the result is the same, even if different matching rules can be applied. For grammars that are not backward confluent, we present a completion algorithm that modifies the grammar to achieve backward confluence. The completion algorithm uses heuristics that do not guarantee language equivalence. Both algorithms are implemented for the model checking tool Attestor.

Contents

1 Introduction

Software development is inherently error-prone. In fact, almost any major software product is nowadays equipped with an update function since shipping bug-free programs is considered infeasible. The more complicated a piece of software is the more likely it is that it contains some kind of programming error. Therefore, a common practice when writing software is to write tests, that check the output of a program for some inputs. It is infeasible to test the program for *all possible* inputs, so in most cases, the developer chooses some set of inputs he deems to be important. While this practice can catch some mistakes early on, the developer may have overlooked some edge cases, for which the program does not work correctly.

A different approach, to avoid unwanted behavior of a program, is *model checking*. Here the developer specifies certain conditions or properties that must be fulfilled by the program. Different kinds of model checking algorithms can check different kinds of properties. For example, it is possible to verify that numerical input values and output values fulfill some arithmetic formula. In contrast to writing tests, the model checking approach checks that the specified properties are fulfilled for *every input*.

This thesis provides improvements for the model checking tool Attestor [1]. Attestor is a model checker for Java programs manipulating data structures. Typical properties within its scope include the absence of null pointer exceptions or that the heap upon program termination consists of a binary tee. It uses graph grammars to abstract different data structures. A developer can add new grammars to model other data structures. Attestor requires that these grammars have a property called *confluence*.

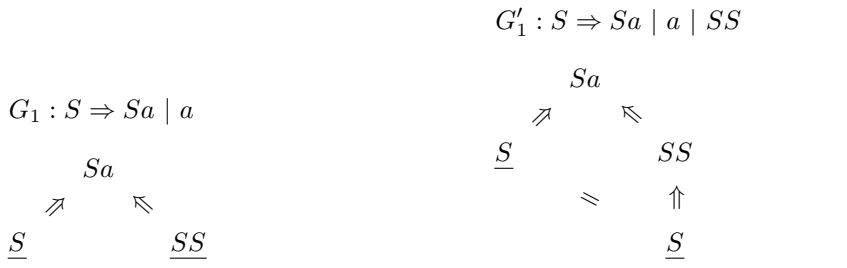
Confluence is a property of formal systems like term rewrite systems, string grammars, and graph grammars, which ensures that all terminating derivation paths starting from the same element must end at the same element. This means that if only the elements at the ends of these derivation chains are of interest, it is not necessary to do backtracking if multiple different derivation steps are possible [11]. Regarding model checking pointer programs, this significantly speeds up the analysis.

In this thesis, we cover confluence for graph grammars, which are more powerful than string grammars and operate on graphs instead of strings. For simplicity, we first explain the advantages of confluence for string grammars, as the basic concept translates to graph grammars. There are two different kinds of confluence: Forward confluence and backward confluence. Backward confluence means that derivation steps of the grammar are applied backward to end up with the same element and for forward confluence, the grammar rules are applied in forward direction. In this thesis, we only consider backward confluence, so we use the term *confluence* to refer to backward confluence.

1.1 Confluence of String Grammars

To better understand what confluence is, consider the example in Figure 1.1. The grammar G_1 in Figure 1.1(a) represents the language $\mathcal{L}_{G_1}(S) = a^+$. The figure also contains a tree, which shows two possible ways that grammar rules can be applied to the word Sa. The rules are applied backward and lead to two different words: S and SS. There is no rule in G, that can be applied backwards to these words. In this case, the two underlined words are said to be in *normal form*. We say that the grammar is not *confluent* because those two words are not equal.

Now consider the modified grammar G_1' in Figure 1.1(b). It contains one additional rule, which means that the word SS is no longer in normal form and applying the new rule leads to S, the same word as in the other derivation path. All backward derivation paths from any word eventually lead to the same word in this grammar, which is why the grammar G_1 is *confluent*. An algorithm



(a) Non confluent grammar with counter-example

(b) Modified confluent grammar with resolved counter-example

Figure 1.1: Confluence in string grammars

that takes a grammar like G_1, which is not confluent, to create a confluent grammar is called a *completion algorithm*.

Having a grammar that is confluent has some benefits. One advantage of a confluent grammar, for example, is that the language inclusion problem becomes decidable under certain conditions. The question if $\mathcal{L}_G(w_1) \subseteq \mathcal{L}_G(w_2)$ holds for a grammar G is undecidable in general. But if G is confluent and w_2 is in normal form it becomes decidable.

As an example consider the following confluent grammar:

$$G_2 : S \Rightarrow Sa \mid b \qquad \mathcal{L}_{G_2}(ba^*)$$

To check, for example, if $\mathcal{L}_{G_2}(Sab) \subseteq \mathcal{L}_{G_2}(S)$ holds, it is only necessary to bring the word Sab to normal form. The word Sab can be abstracted in two steps $Sab \Leftarrow Sb \Leftarrow SS$ and there is no other rule in G_2 that can be applied to SS, so SS is the normal form of Sab. The word SS is then compared with S and because $S \neq SS$, it means that $\mathcal{L}_{G_2}(Sab) \not\subseteq \mathcal{L}_{G_2}(S)$. This conclusion is only valid because the grammar G_2 is confluent.

1.2 Confluence in Attestor

As mentioned before, the model checking tool Attestor uses graph grammars to abstract data structures. These grammars allow Attestor to determine if a program state contains an invalid data structure. For example, a program that takes a binary tree as input and returns a modified tree can be analyzed with Attestor to verify that for *all possible* inputs the program *always* returns a valid binary tree. When analyzing loops of a program Attestor uses a language inclusion check to determine if all cases of the next loop iteration are already covered. This is one reason, why Attestor requires that its grammars are backward confluent. Part of this thesis is the implementation of a confluence check, which is integrated into Attestor so it is no longer necessary to manually verify that the confluence property is fulfilled. This is very useful, as it becomes increasingly difficult to verify if a grammar is confluent, the larger the grammar is. With our implementation, we were able to reveal that a grammar, that is used in Attestor to model doubly linked lists is *not confluent*.

In case a grammar is not confluent, we also provide an implementation of a completion algorithm, which automatically modifies the grammar to achieve confluence. While this process is not guaranteed to succeed for every grammar, it can still be helpful to a user when creating a grammar. We implemented PDF output, that shows the modified grammar and eventual remaining issues that prevent the grammar from being confluent. So if it is not possible to create a confluent grammar automatically, this output can be used to better understand the issues with the initial grammar and modify the grammar by hand.

2 Preliminaries

In this chapter, we formally introduce the definitions and important properties related to hypergraphs and hyperedge replacement grammars (HRGs) that are used in this thesis.

2.1 Hypergraphs

Graphs are structures that are very useful in the domain of computer science. They can be used to model communication networks, relations between people in social networks and many other fields. A graph consists of a set of nodes that are connected by edges. Each edge connects two nodes and in a directed graph, the order in which nodes are connected is distinguished. A hypergraph is an extension to the graph concept, which introduces the concept of hyperedges that can not only connect to two nodes but an arbitrary number of nodes. An example of a hypergraph is shown in Figure 2.1. Here, the nodes are represented by circles and the hyperedges are shown as rectangles. Lines between hyperedges and nodes indicate what nodes are attached to a hyperedge. Each of those lines has a number in the middle to indicate the order of the attached nodes. Each hyperedge is assigned a label from a ranked alphabet Σ. Ranked means that an integer is assigned to each label of Σ by using a ranking function $rk : \Sigma \to \mathbb{N}$. The rank of a hyperedge specifies the number of nodes it can connect to. This means that in the given example the edge with label A has rank 2 and the edge with label B has rank 3. There can be multiple hyperedges with the same label, so we use the identifier in the top left of the hyperedge (n and m) to distinguish them. We call the connections between a hyperedge and the connected nodes *tentacles*. The order in which nodes are connected to the hyperedges is important and we indicate this order by showing numbers at the tentacles from 1 up to the rank of the specific hyperedge. It is allowed that the same hyperedge connects to the same node multiple times with different tentacles.

In the example hypergraph, we also see that some nodes are shown with a double stroke. This indicates that those nodes are so-called external nodes. The purpose of these external nodes is explained in more detail in Section 2.2 but for now it suffices to know that external nodes are a special sequence of the nodes of the hypergraph. The formal definition of a hypergraph is shown in the following:

Definition 1 (Hypergraph [4])**.** Let Σ be a finite ranked alphabet, where $rk : \Sigma \to \mathbb{N}_0$ assigns to each symbol $X \in \Sigma$ its rank $rk(X)$. A *(labeled) hypergraph over* Σ is a tuple

$$H = (V, E, att, lab, ext),$$

where V is a finite set of vertices, E a finite set of hyperedges, $att : E \to V^*$ maps each hyperedge to a sequence of attached vertices, $lab : E \to \Sigma$ is a hyperedge-labeling function, and $ext \in V^*$ a (possibly empty) sequence of pairwise distinct external vertices. If $ext = v_1...v_n$, then $ext(i) = v_i$ references the i^{th} element (where $i \in [1, n]$). For every $e \in E$, we require $|att(e)| = rk(lab(e))$, and we let $rk(e) = rk(lab(e))$.

The set of all hypergraphs over Σ is denoted by HG_Σ.

In the example hypergraph from Figure 2.1, we have $V = \{1, 2, 3\}$, $E = \{n, m\}$, $\Sigma = \{A, B\}$, $att := [n \mapsto (1, 2), m \mapsto (1, 2, 3)]$, $lab := [n \mapsto A, m \mapsto B]$, and $ext := [1, 2]$. Note that in some depictions of hypergraphs we leave out the identifiers of nodes and hyperedges if the identification of those nodes and edges is not of importance. We always assign external nodes identifiers that correspond to the index in the sequence of external nodes.

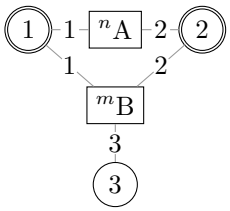


Figure 2.1: A Hypergraph

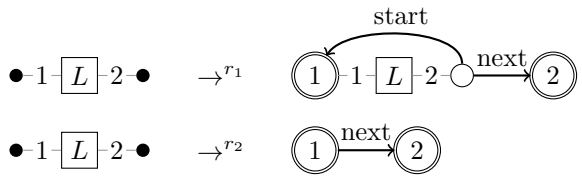


Figure 2.2: Example HRG with two rules

A tentacle is defined as a tuple containing the edge and an integer indicating which node connection it refers to.

Definition 2 (Tentacle [3])**.** Given a hypergraph $H \in HG_\Sigma$, an edge $e \in E_H$ with $lab(e) = a$ and $1 \leq i \leq rk(a)$. We call the tuple (e, i) a *tentacle*.

A tentacle can be used to refer to a certain node relative to a specific hyperedge.

2.2 Hyperedge Replacement Grammars

String grammars are used to represent an infinite set of words using a finite representation. Similarly, a grammar can be defined that uses graphs instead of words to represent an infinite set of graphs. The kind of grammar that is used in this thesis is called hyperedge replacement grammar (HRG) and similar to string grammars it uses a set of rules that have a left-hand side (LHS) and a right-hand side (RHS). Each rule specifies how a hypergraph can be modified by removing a hyperedge (the LHS) and replacing it with a different graph (the RHS). One rule application only allows the removal of a single hyperedge and the inserted hypergraph is only allowed to connect to nodes that were connected to the hyperedge that is removed in this step.

Definition 3 (Hyperedge replacement grammar [4])**.** *An HRG* G over an alphabet Σ_N is a finite set of production rules of the form $X \to H$, where $X \in N$, $H \in \mathrm{HG}_{\Sigma_N}$, and $|ext_H| = rk(X)$. The set of all HRGs over Σ_N is denoted by HRG_{Σ_N}.

Definition 3 shows what those rules need to adhere to for a valid HRG. An important detail is that the number of external nodes in the RHS must to correspond to the rank of the LHS. The reason for this requirement is that when a hyperedge is replaced by the RHS of a rule the external nodes are used to define how the RHS is inserted into the graph from which the hyperedge is removed. The exact process of this hyperedge replacement is described in Definition 4. Here we use a set of labels Σ_N, which contains a set of labels N for so-called *nonterminal edges*. Only edges with a nonterminal label are allowed to be an LHS. All other edges are called *terminal edges*.

In Figure 2.2 an example grammar that contains two rules is shown. All rules have the nonterminal edge L as their LHS, which is shown left of the arrow $\to$. According to the definition of HRGs, the LHS is not a graph, but just a nonterminal. It does not contain nodes, but we still show the tentacles of the edge so the rank can be seen easily. The ends of these tentacles are filled out black circles to indicate that they are not in fact nodes of a hypergraph. To distinguish terminal and

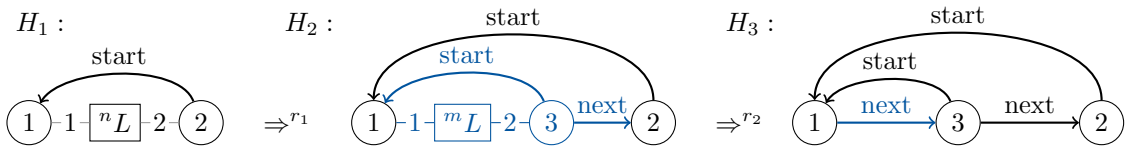


Figure 2.3: Example of two consecutive HRG derivation steps

nonterminal edges, the terminal edges are shown as arrows like in an ordinary directed graph. This is possible, as all HRGs we consider only have terminal edges of rank 2.

A rule of an HRG can be applied to a nonterminal edge of a hypergraph according to Definition 4. In this definition the symbol $\restriction$ is used to restrict the function on the left side of the symbol to input values of the set on the right side of the symbol. Also, we use $att_H(e)(i)$ to refer to the i^{th} node attached to edge e in hypergraph H. We use $[ext_K]$ to refer to the *set* of external nodes in hypergraph K based on the *sequence* of external nodes ext_K.

Definition 4 (Hyperedge replacement [4])**.** Let $H, K \in \mathrm{HG}_{\Sigma_N}$ and $e \in E_H$ a nonterminal edge with $rk(e) = |ext_K|$. W.l.o.g. let $V_H \cap V_K = E_H \cap E_K = \emptyset$. The *substitution of e by K*, denoted $H[K/e]$, is the hypergraph $H' \in \mathrm{HG}_{\Sigma_N}$ defined by:

$$
\begin{aligned}
V_{H'} &= V_H \cup (V_K \setminus [ext_K]) \\
E_{H'} &= (E_H \setminus \{e\}) \cup E_K \\
lab_{H'} &= (lab_H \restriction (E_H \setminus \{e\})) \cup lab_K \\
ext_{H'} &= ext_H \\
att_{H'} &= att_H \restriction (E_H \setminus \{e\}) \cup (mod \circ att_K)
\end{aligned}
$$

where $mod = \mathrm{id}_{V_{H'}} \cup \{ext_K(1) \mapsto att_H(e)(1), ..., ext_K(rk(e)) \mapsto att_H(e)(rk(e))\}$.

When edge e is replaced in graph H in the definition, the set $V_{H'}$ contains all nodes from H and K, except the external nodes in K because those nodes are mapped onto nodes that already exist in H. The sets of edges are united, excluding the one edge that was removed and the edge is also removed from the *lab* and *att* function of the resulting graph H'. Additionally, the *att* function is modified so that the external nodes in K correctly connect to the nodes from H, for which the definition uses the newly introduced *mod* function.

Using this edge replacement mechanism, it is then possible to define a relation that indicates if one hypergraph can be derived from another hypergraph. In the definition we refer to an isomorphism between two hypergraphs. For an isomorphism between two hypergraphs there must be a mapping between the nodes and edges, so that the sequence of external nodes, labels of the edges and connections between nodes and edges is maintained. The definition of this relation is shown in the following:

Definition 5 (HRG derivation [4])**.** Let $G \in \mathrm{HRG}_{\Sigma_N}, H, H' \in \mathrm{HG}_{\Sigma_N}$, $p = X \to K \in G$ and $e \in E_H$ with $lab_H(e) = X$. *H derives H'* by p, denoted $H \Rightarrow_{e,p} H'$ iff H' is isomorphic to $H[K/e]$. Let $H \Rightarrow_G H'$ if $H \Rightarrow_{e,p} H'$ for some $e \in E_H$ and $p \in G$. If G is clear from the context, we write $H \Rightarrow H'$ instead. Further, we let $\Rightarrow^*$ denote the reflexive and transitive closure of $\Rightarrow$.

In Figure 2.3, an example of two derivation steps using the grammar from Figure 2.2 is shown. The hypergraph H_1 in the left of the figure is just an arbitrary start graph to which in a first step rule r_1 is applied to edge n. The nonterminal edge n is removed and the connected tentacles indicate that node 1 corresponds to external node 1 of the RHS of r_1 and node 2 corresponds to external node 2 in the RHS of r_1. The RHS is then added to H_1 such that the corresponding external nodes match. The result of this operation is shown in the figure as hypergraph H_2. Here

all newly added elements are shown in blue. Note that the nonterminal L now has a different id m, as it is not the same edge from H_1, but was newly inserted because it is contained in the RHS of the applied rule. All non-external nodes from the RHS must be added to the graph as new nodes. This is the reason H_2 contains a new node with id 3. The id 3 is chosen arbitrarily just to distinguish the nodes, but it must be unique in the hypergraph.

Lastly, rule r_2 is applied to the nonterminal edge m of H_2. Here the tentacles of edge m indicate that node 1 corresponds to external node 1 of the RHS and node 3 corresponds to external node 2 of the RHS, as the tentacle with index 2 is connected to node 2. So in the derivation step, the nonterminal edge m is removed and the RHS is inserted. In the RHS of rule r_2, all nodes are external nodes, so no new nodes are added and just the terminal edge with label *next* is added. The result is shown as H_3 with the new edge marked in blue.

We refer to the language of a hypergraph with respect to some HRG as the set of all hypergraphs without nonterminals that can be derived from the given hypergraph. We use HG_Σ to exclude hypergraphs with nonterminal edges that are included in HG_{Σ_N}.

Definition 6 (Language of HRG [4])**.** The *language* generated from $H \in \mathrm{HG}_{\Sigma_N}$ with respect to $G \in \mathrm{HRG}_{\Sigma_N}$ is defined by $\mathcal{L}_G(H) = \{K \in \mathrm{HG}_\Sigma \mid H \Rightarrow^* K\}$.

We also use the term *handle* (Definition 7) to refer to a hypergraph that contains a single edge with a specific nonterminal label that is also connected to a number of nodes corresponding to the rank of the edge. A nonterminal edge in itself does not constitute a valid hypergraph and the handle notation allows to easily refer to a simple valid hypergraph with one nonterminal.

Definition 7 (Handle [4])**.** Given $X \in N$ with $rk(X) = n$, an X-*handle* is the hypergraph $X^\bullet = (\{v_1, ..., v_n\}, \{e\}, [e \mapsto v_1...v_n], [e \mapsto X], v_1...v_n) \in \mathrm{HG}_{\Sigma_N}$.

2.2.1 Backward Rule Application

With an HRG, rules cannot only be applied in the normal forward direction but also backward. While it would be possible to define the rule application implicitly by using the definition for forward rule application, i.e. $H \Leftarrow K$ iff $K \Rightarrow H$, it is beneficial to introduce a separate definition that more closely resembles how an implementation for applying rules backward works.

When a rule is applied in forwards direction a hyperedge is replaced by a hypergraph, so when applying a rule backward, a hypergraph has to be replaced by a hyperedge. To determine if it is possible to apply a specific rule in reverse for some hypergraph H, the RHS graph of the rule R must be a subgraph of H. The matched graph in H is then removed, while nodes that match an external node in R are kept. Those nodes are then connected to a newly added hyperedge with the label of the LHS.

We call the mapping of one hypergraph to a subgraph of another graph an embedding.

Definition 8 (Embedding [4])**.** Given $R, H \in \mathrm{HC}_{\Sigma_N}$, an *embedding emb* of R in H is a pair of mappings $emb_V : V_R \to V_H$ and $emb_E : E_R \to E_H$ with the following properties:

$$
\begin{aligned}
emb_V(v) &\notin [ext_H] && \forall v \in V_R \setminus [ext_I] \\
emb_V(v) &\neq emb_V(v') && \forall v \in V_R, v' \in V_R \setminus [ext_R] \\
& && \text{with } v \neq v' \\
emb_E(e) &\neq emb_E(e') && \forall e, e' \in E_R \text{ with } e \neq e' \\
lab_R(e) &= lab_H(emb_E(e)) && \forall e \in E_R \\
emb_V(att_R(e)) &= att_H(emb_E(e)) && \forall e \in E_R \\
e \notin emb_E(E_R) &\Rightarrow [att_H(e)] \cap emb_V(V_R \setminus [ext_R]) = \emptyset && \forall e \in E_H.
\end{aligned}
$$

We denote the set of all embeddings of I in H with $Emb(I, H)$.

For a valid embedding of R in H, all nodes that are not external in I must not be external in H. This ensures that external nodes are never deleted by applying a rule backward. It is also forbidden that in general one node or edge in I maps to two different nodes in H. An exception to this is that *external nodes* of R *can* map to the same node in H. This is necessary to apply a rule backward where the involved LHS connects with two tentacles to the same node after the backward rule application. The last condition in the definition ensures that no edge in H is connected to a node that would be deleted during backward rule application even though the edge would not be deleted. If this condition would not be fulfilled and the embedded RHS is removed the resulting hypergraph would contain a hyperedge that is not connected to the correct number of nodes, which is not allowed. This condition is therefore sometimes called the *dangling edge condition*.

In Definition 9 the actual replacement of the matching RHS with the LHS is shown. All nodes except the external nodes from the RHS are removed while a new edge with the correct label and node attachment is added.

Definition 9 (Hypergraph replacement [4])**.** Given $G \in \mathrm{DSG}_{\Sigma_N}, R, H \in \mathrm{HC}_{\Sigma_N}, emb \in Emb(R, H)$ and $X \in N$ with $rk(X) = |ext_R|$, replacing R in H results in

$$replace(R, H, emb, X) = K,$$

where

$$\begin{aligned} V_K &= V_H \setminus emb_V(V_R \setminus [ext_R]) \\ E_K &= (E_H \setminus emb_E(E_R)) \uplus \{e\} \\ lab_K &= (lab_H \restriction E_K) \cup \{e \mapsto X\} \\ att_K &= (att_H \restriction E_K) \cup \{e \mapsto emb_V(ext_R)\} \\ ext_K &= ext_H. \end{aligned}$$

According to Lemma 1 below, the backward rule application using embeddings and the *replace* function is compatible with the forward derivation from Definition 4. So a graph H can only be derived from a graph K by Definition 4 iff there is an embedding that allows the backward rule application from graph H to graph K.

Lemma 1 (Compatibility of forward and backward derivation [4])**.** Let $G \in DSG_{\Sigma_N}$ and $H, I, K \in HC_{\Sigma_N}$.

Then: $K \Rightarrow H$ iff there exists a rule $X \to I \in G$ and embedding $emb \in Emb(I, H)$ such that $K = replace(I, H, emb, X)$

We refer to the forward rule application as *partial concretization* and the backward rule application as *abstraction*. The reason for this is that the forward application of rules *restricts* the language, while the backward application *increases* the language of the graph.

We call a hypergraph *fully abstracted* if there is no rule in the grammar that can be backward applied to the graph. According to Definition 10, for some hypergraph H, the function $fullAbstr_G(H)$ contains a set of all hypergraphs from which H can be derived and which are fully abstracted.

Definition 10 (Fully abstracted hypergraphs [4])**.** Given $G \in \mathrm{HRG}_{\Sigma_N}$ and $H \in \mathrm{HG}_{\Sigma_N}$, the set $fullAbstr_G(H)$ of *fully abstracted* hypergraphs of H is defined as:

$$fullAbstr_G(H) = \{K \in \mathrm{HG}_{\Sigma_N} | K \Rightarrow^* H \wedge \forall X \to R \in G : Emb(R, K) = \emptyset\}.$$

2.2.2 Collapsed Rules

From the definition of rule applications, it follows that multiple external nodes can be mapped to a single node in a hypergraph when a rule is applied. This is realized in Attestor by so-called *collapsed rules*. In Attestor a grammar contains a set of *original rules*, which do not allow multiple

external nodes to map to the same hypergraph node. This means a base rule cannot be applied to a nonterminal where two tentacles are attached to the same node. A *collapsed rule* is based on an original rule but allows for one specific combination of overlap of external nodes. When a grammar is initialized in Attestor, all possible *collapsed rules* are computed and added to the grammar.

2.2.3 Heap Configurations

The confluence detection and completion algorithms that are presented in this thesis are developed to improve the model checking tool Attestor [1]. This tool uses a special subset of hypergraphs called *heap configurations*. The purpose of a heap configuration is to represent the state of the heap at a specific instruction during the execution of a program. Attestor analyzes Java bytecode, which is object-oriented. The different nodes in a heap configuration represent the different allocated objects of the program. A node can also represent a `NULL` value.

An important difference to the previously introduced hypergraphs is the division of the set of edge labels Σ into three different categories. A heap configuration can contain *variable edges* (Var_Σ), *selector edges* (Sel_Σ) and other *nonterminals* (N). We use the symbol $\uplus$ to indicate the union of two sets where we assume that the two sets are disjoint.

The variable edge labels each represent a different variable on the stack during program execution. All variable edges must have a rank equal to 1 and they are connected to the node that represents the object that the variable is currently referencing. The definition enforces that there cannot exist multiple variable edges with the same label as a variable can only reference a single object.

The heap configuration also introduces the concept of *selector edges*, which are hyperedges of rank 2. In object-oriented programming languages, each object can reference other objects using fields. These references are represented by selector edges. Different labels of the selector edges indicate the different kinds of fields that an object might have. For a node, there cannot exist multiple outgoing selector edges with the same label because a field of an object can only reference *a single* object. All variable edges and selector edges are terminal edges and therefore cannot appear in an LHS.

Only the other class of hyperedge labels, the *nonterminals*, can be used as LHS's in an HRG. The usage of HRGs in Attestor allows the representation of an infinite set of heap configurations that all share some kind of common structure. A nonterminal here can, for example, be used to represent all possible linked-lists of arbitrary length.

Definition 11 (Heap configuration [4])**.** A *heap configuration* is a hypergraph $H \in \mathrm{HG}_{\Sigma_N}$ with $H = (V, E, att, lab, ext)$ where

- $\Sigma_N = \Sigma \uplus N$ and $\Sigma = Var_\Sigma \uplus Sel_\Sigma$,
- $rk(Var_\Sigma) = \{1\}$ and $rk(Sel_\Sigma) = \{2\}$,
- for every $x \in Var_\Sigma, |\{e \in E \mid lab(e) = x\}| \leq 1$,
- for every $v \in V$ and $s \in Sel_\Sigma$, $|\{e \in E \mid lab(e) = s, \exists w \in V : att(e) = vw\}| \leq 1$, and
- $ext = \epsilon$, where ϵ denotes the empty sequence.

The set of all heap configurations is denoted by HC_Σ.

The example in Figure 2.4 shows the connection between a program state and the corresponding heap configuration. The code on the left creates two new objects and assigns them to the variables a and b. Then the field s of the object a is set to reference object b. In the heap configuration shown at the right in the figure, the objects a and b are represented by two nodes. The variable edges with labels a and b specify which object the two variables reference. To distinguish selector edges from the other edges, we show them like edges in a classic directed graph and do not use the hyperedge notation. The source node of the selector edge is connected to the 1^{st} tentacle and the target node is connected to the 2^{nd} tentacle.

```
A a = new A();
B b = new B();
a.s = b;
```

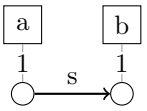


Figure 2.4: Connection between Java code and a heap configuration

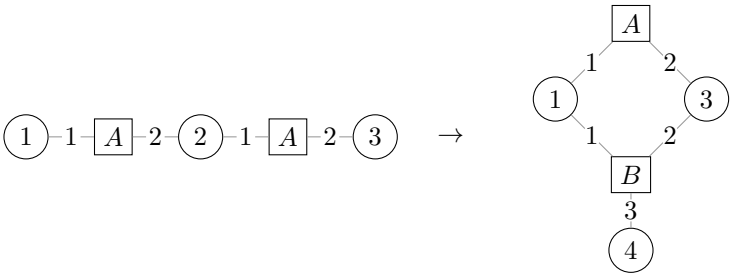


Figure 2.5: Example rule of general graph transformation system

2.3 Confluence

Confluence is a property of rewriting systems like term rewriting systems, string grammars, or in our case HRGs. The general idea of confluence is that for all elements for which two divergent paths of rewrites exists, it is always possible to end up at the same element again by applying additional consecutive rewrite steps after the two divergent paths of rewrites.

We first give an introduction to confluence in general graph transformations, as most theorems for confluence in HRGs are based on theorems for this more general system. Later in this section, confluence, or to be more precise *backward confluence*, is introduced formally for HRGs.

2.3.1 Confluence in General Graph Transformation Systems

In HRGs, rules are restricted by the fact that an LHS must be a single edge. General graph transformation systems allow arbitrary graphs as LHS. Every HRG can be represented as a general graph transformation system, while the inverse is not true. As HRGs are a special case of general graph transformation systems, certain theorems that apply to general graph transformation systems can be adapted to HRGs.

The less restrictive format of rules in general graph transformation systems makes them more expressive because a rule can add *and* remove multiple nodes and edges. A rule in an HRG cannot remove nodes and can just remove a single edge. In general graph transformation systems, it is also possible to reverse every rule, which means a new grammar can be created, where the derivation relation is reversed. The same is not possible for HRGs, as the LHS cannot be an arbitrary hypergraph like the RHS. Figure 2.5 shows an example of a valid rule of a general graph transformation system, which cannot be represented in an HRG. The numbers inside the nodes indicate which node of the LHS corresponds to which node of the RHS. The rule deletes node 2 and a new node with the number 4 is added.

An important theoretical result for general graph transformation systems from [10], is that it is undecidable if a general graph transformation system is confluent. There is a special subset of general graph transformation systems called coverable systems, which is introduced in [12]. For terminating, coverable systems confluence is decidable. The term *terminating* in this context means that there are only finitely many consecutive rule applications possible for a hypergraph. A graph transformation system is terminating, for example, if every rule only removes edges or nodes from the graph. As the start graph has a finite size at the beginning, this means that at some point after multiple rule applications it is no longer possible to apply any more rules.

2.3.2 Backward Confluence in HRGs

For HRGs, we are interested in whether a grammar is *backward confluent* or not. It is equivalent to confluence of a general graph transformation system that represents the HRG, but where all rules are reversed. Because for HRGs the reversal of rules is not possible in general, we need the additional property of *backward confluence*. Determining if an HRG is forward confluent is trivial because of context-freeness [13]. Therefore in the remainder of the thesis, we refer to backward confluence of HRGs, by just using the term *confluence*.

Definition 12 shows how backward confluence is defined for HRGs. The definition requires so-called *increasing grammars*, which means that RHS's must either contain terminal edges or the number of edges or nodes must be larger than the LHS handle. This ensures that there are no infinite paths of backward rule applications for any hypergraph, i.e. the underlying graph transformation system is terminating.

Definition 12 (Backward confluence [3])**.** An increasing grammar $G \in \mathrm{HRG}_\Sigma$ is *backward confluent* iff for any $H \in \mathrm{HG}_\Sigma$ with $H \Leftarrow_G K_1$ and $H \Leftarrow_G K_2$ there is a $K \in \mathrm{HG}_\Sigma$ with $K_1 \Leftarrow^* K$ and $K_2 \Leftarrow^* K$.

Critical Pairs

Critical pairs can be used to determine if a grammar is confluent. If a grammar is *not confluent*, then there must always be some graph where there are two possible backward rule applications possible. The idea of *critical pairs* is to only look at minimal graphs were those splits occur. By *minimal*, we mean that the graph *only* contains nodes and edges that are necessary so the two rules that lead to a violation of confluence can be applied. A splitting point provides *no violation* of confluence if the two rules can be applied immediately one after the other. If the two rule applications do not interfere with each other, we call them independent, according to Definition 13. Two rules are not independent if one rule application requires the presence of a specific node or edge, which is removed by the other rule application.

Definition 13 (Independence [3])**.** Given $H, R_1, R_2 \in \mathrm{HG}_\Sigma$ two embeddings emb_1 and emb_2 with $emb_1 \in Emb(R_1, H)$ and $emb_2 \in Emb(R_2, H)$ are *independent* iff $emb_1(V_{R_1} \setminus ext_{R_1}) \cap emb_2(V_{R_2} \setminus ext_{R_2}) = \emptyset$ and $emb_1(E_{R_1}) \cap emb_2(E_{R_2}) = \emptyset$.

Using the definition of independence, it is possible to then define what constitutes a valid critical pair, according to Definition 14. The critical pair consists of the common graph H and the two backward rule applications. Here it is not only important *which* rules of the grammar are used, but also *what part* of H they apply to. There can be two different critical pairs, which consist of the same graph H and which both use the same pair of rules, but the embeddings of the RHS's are different.

Definition 14 (Critical Pair [3])**.** Given $G \in \mathrm{HRG}_\Sigma$, a *critical pair* consists of a graph and two backward rule applications $(K_1 \Rightarrow_{p_1} H \Leftarrow_{p_2} K_2)$ with $p_i = X_i \to R_i$ and the embeddings $emb_i \in Emb(R_i, H)$ for which $K_i = replace(R_i, H, emb_i, X_i)$ with $i \in \{1, 2\}$. A valid critical pair must fulfill the following two properties:

- $emb_1(V_{K_1}) \cup emb_2(V_{K_2}) = V_H$ and $emb_1(E_{K_1}) \cup emb_2(E_{K_2}) = E_H$ (Minimality of H)
- emb_1 and emb_2 are not independent.

The minimality of H implies that there is only a finite amount of critical pairs. For every combination of two rules, the graph H of the critical pair can only contain the nodes and edges from both RHS's, which might overlap. In this thesis, we refer to the graph H in the critical pair as the *joint graph*.

In Figure 2.6 an example of a critical pair is shown. The critical pair is created from rules r_1 and r_2 from the grammar of Figure 2.2. The joint graph is shown at the top and the two abstractions

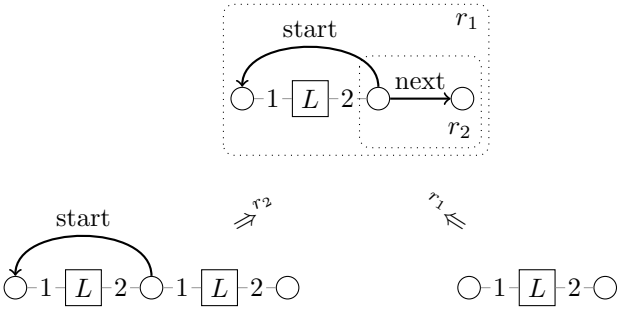


Figure 2.6: Critical pair example with embeddings marked in joint graph

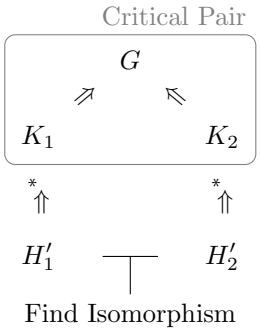


Figure 2.7: Critical pair joinability

are shown below. In the joint graph, the dotted lines also indicate how the two RHS's of rule r_1 and r_2 overlap. Both abstracted graphs are fully abstracted because there is no RHS in the grammar that is contained in any of the two graphs.

For each critical pair, it is possible to compute its *joinability* to determine if the critical pair is a counter-example for the confluence of the given grammar. For a critical pair to be joinable, there must be two abstraction paths from the two abstracted graphs that lead to two isomorphic graphs.

Definition 15 (Joinability [3])**.** A critical pair $K_1 \Rightarrow_{p_1} H \Leftarrow_{p_2} K_2$ is joinable iff there are two possible abstraction paths $K_1 \Leftarrow^* H_1'$ and $K_2 \Leftarrow^* H_2'$ where H_1' and H_2' are isomorphic.

In Figure 2.7 this is further illustrated. At the top, the critical pair is shown which includes the graphs C_1 and C_2. These graphs are fully abstracted to the graphs H_1' and H_2' which are then checked for an isomorphism.

If there is no isomorphism between H_1' and H_2', then the critical pair is a counter-example for confluence and it is not necessary to check any other critical pairs. However, even if all critical pairs are joinable, this would not directly imply that the grammar is confluent. The reason for this is the minimality condition. If an additional edge is added to the joint graph of the critical pair, which is not deleted during the two abstraction paths, it would add additional constraints on the isomorphism for the joinability. There is a stricter notion of joinability, called *strong joinability*. Here the isomorphism between H_1' and H_2' has to fulfill additional conditions. The nodes which are not deleted during the two abstraction paths, called common nodes, must match exactly in the isomorphism.

Definition 16 (Strong Joinability [3])**.** A critical pair $K_1 \Rightarrow_{p_1} H \Leftarrow_{p_2} K_2$ is strongly joinable iff there are two possible abstraction paths $K_1 \Leftarrow^* H_1'$ and $K_2 \Leftarrow^* H_2'$ and an isomorphism $iso : (V_{H_1'} \cup E_{H_1'}) \to (V_{H_2'} \cup E_{H_2'})$ satisfying $iso \restriction V_{H_2'} = \mathrm{id}_{V_{H_1'} \cap V_{H_2'}}$.

Note that according to the definition of the forward and backward rule application, nodes that are not added or removed during the rule application are *equal* in both graphs. Therefore the common nodes of H_1' and H_2', in the definition above, are exactly $V_{H_1'} \cap V_{H_2'}$. So by restricting the isomorphism *iso* to $V_{H_2'}$, its domain is exactly the set of common nodes. So if the restricted isomorphism is equal to the identity function over the common nodes, then all the common nodes are mapped correctly.

If a critical pair is strongly joinable, it is not possible that an additional edge in the joint graph that is not removed during the two abstraction paths adds any additional constraints that contradict the isomorphism for the strong joinability. As the edge is not deleted in both paths it must be connected to the common nodes, which are maintained by the isomorphism. Based on this idea, it is possible to show that a grammar is confluent if all its critical pairs are strongly joinable [10].

We then introduce the term *weakly joinable* according to the following:

Definition 17 (Weak Joinability)**.** A critical pair is *weakly joinable* if it is joinable, but not strongly joinable.

Decidability of Backward Confluence

In the previous section, we introduced the concept of joinability and strong joinability. A grammar that has a critical pair, which is not joinable cannot be confluent. Furthermore, a grammar where all critical pairs are strongly joinable is guaranteed to be confluent. For general graph transformation systems confluence is not decidable if there are any *weakly joinable* critical pairs [10]. Howerever in [12], so-called coverable hypergraph transformation systems are introduced, for which all critical pairs must be strongly joinable or the grammar is not confluent. So confluence is decidable for these systems. The general idea of coverable hypergraph transformation systems is that there is a way to extend the joint graphs of all critical pairs such that any valid isomorphism must enforce that the common nodes map to themselves. It turns out that HRGs are all coverable hypergraph transformation systems because our hypergraph definition includes external nodes [4]. In an isomorphism between two hypergraphs, external nodes that have the same index must map to each other. So by declaring the common nodes external, the common nodes must match accordingly for any valid isomorphism. This means that HRGs are coverable, which makes confluence is decidable.

Note that in the special case of heap configurations there are no external nodes allowed, according to their definition. However, as the heap configurations are used to represent the state of arbitrary programs, they could include any number of variables. Variable edges can therefore be used instead of external nodes to distinguish the common nodes. A different variable edge is attached to each of the common nodes, which has the same effect that only isomorphisms that correctly map the common nodes are valid.

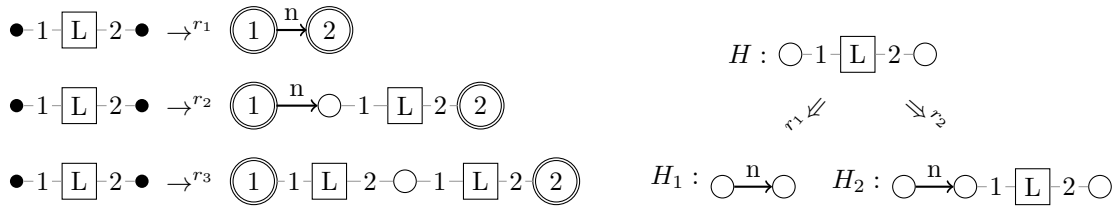
Using these insights it is possible to adapt the critical pair lemma from [10] for HRGs. The critical pair lemma shown in the following states that confluence is fulfilled only if all critical pairs are strongly joinable.

Lemma 2 (Critical pair lemma for HRGs)**.** An increasing HRG is *confluent* iff all its critical pairs are *strongly joinable*.

This property must be fulfilled for all grammars that are used in Attestor.

2.4 Data Structure Grammars

As described in Section 2.2.3, heap configurations are only allowed to have a single outgoing selector of the same label per node. While this property might hold for some hypergraph, the application of a grammar rule could lead to a violation of this property. To avoid problems like this, the concept of *data structure grammars* is introduced. For these grammars, all possible derivations from a handle graph lead to valid heap configurations.



(a) Local concretizable grammar (b) Concretization example

Figure 2.8: Locally concretizable grammar with concretization example

Definition 18 (Data structure grammar [4])**.** $G \in \mathrm{HRG}_{\Sigma_N}$ is a data structure grammar (DSG) over Σ_N if $\forall X \in N : \mathcal{L}(X^\bullet) \subseteq \mathrm{HC}_{Sel_\Sigma}$.

Note that when a grammar rule is applied to an arbitrary hypergraph and not a handle, then the resulting hypergraph might not be a valid heap configuration. It is decidable if a given HRG is a data structure grammar [4]. Later, in Section 5.1.3 we present an algorithm that determines if a grammar is a data structure grammar.

Attestor is only able to represent heap configurations so all HRGs used with it must be data structure grammars. In the computation of collapsed rules, Attestor ensures that the grammar is still a data structure grammar by preventing the merge of nodes in a rule where both nodes can get outgoing selectors.

2.5 Local Concretizability

Attestor uses nonterminals in heap configurations as a way to abstract data structures and then executes program operations on these abstract states. For this, it is necessary to be able to concretize the abstract heap configurations so the heap configuration contains selector edges that are required by the program operation. Local concretizable grammars have the property that for any node in a hypergraph *at most one* rule application for each connected nonterminal is required to create outgoing selectors with *all possible* labels that can be derived at that node. All grammar rules, that do not immediately create the necessary outgoing selectors, can be omitted without altering the language.

In the definition of local concretizability (Definition 19), we refer to the subset of grammar rules with X as LHS by G^X and the inverse, i.e. the grammar with all rules that *do not* have X as LHS by $\overline{G^X}$. The set $type(X, i)$ denotes the set of all labels of outgoing selectors that can be derived at the i^{th} tentacle of X either immediately, or by successive application of rules. The grammars $G_{(X,i)}$ can only contain rules that *immediately* add all required outgoing selectors to satisfy the second condition in the definition. The first condition ensures that leaving out the other rules does not modify the language of the grammar.

Definition 19 (Local concretizability [4])**.** $G \in \mathrm{DSG}_{\Sigma_N}$ is *locally concretizable* if for all $X \in N$ there exist grammars $G_{(X,1)}, ..., G_{(X,rk(X))} \subseteq G^X$ such that $\forall i \in [1, rk(X)]$:

1. $\mathcal{L}_{G_{(X,i)} \cup \overline{G^X}}(X^\bullet) = \mathcal{L}_G(X^\bullet)$, and
2. $\forall a \in type(X, i), p \in G_{(X,i)}$: The i^{th} external node in the RHS of p has an outgoing selector with label a.

In Figure 2.8(a) an example of a locally concretizable grammar is shown. To understand why this grammar is local concretizable, consider the graph H in Figure 2.8(b). Notice that the grammar can only create an outgoing selector with label n at the left node. Outgoing selectors are *never* created at the right node. It is possible to concretize the nonterminal edge L in H by all three rules

in the grammar, but only rules r_1 and r_2 *immediately* create the outgoing selector edge. However, if rule r_3 is eliminated from the grammar the language $\mathcal{L}(X^\bullet)$ does not change. So in this example, the two grammars $G_{(L,1)}$ and $G_{(L,2)}$ would both contain only rule r_1 and r_2.

3 Related Work

This chapter presents research, that is related to the topic of this thesis. First, we present a completion algorithm for a different formal system, called *term rewrite system*. We give a short explanation of how those systems work and how they relate to hyperedge replacement grammars (HRGs).

Additionally, we discuss the Attributed Graph Grammar System (AGG) tool, which contains an implementation to compute the critical pairs of a graph grammar.

3.1 Completion of Term Rewriting Systems

When speaking of confluence, completion is an algorithm that can transform a non-confluent system into a confluent one. In this section, we want to present the *Knuth-Bendix completion* algorithm, which was designed for term-rewriting systems. Therefore we give a short informal introduction to term-rewriting systems to present the fundamental idea of Knuth-Bendix completion.

Term rewriting systems are similar to grammars. Instead of arbitrary words, a term rewriting system works with terms. An example of a term rewriting system is shown in the following [2]:

$$\begin{aligned} x + 0 &\to^{r1} x \\ s(x + y) &\to^{r2} x + s(y) \end{aligned}$$

In this term rewriting system, the term $s(x + y + 0)$ can be rewritten in the following way: $s(x + y + 0) \to^{r1} s(x + y) \to^{r2} x + s(y)$. A difference to string grammars is, that left-hand sides (LHS's) are not matched character by character, but variables like x and y can be renamed to other variables or constants. For example, $s(y + 0)$ can be rewritten to $y + s(0)$ by using rule $r2$.

It is possible to define a special strict order on all possible terms, called *Knuth-Bendix order* which is later used by the completion algorithm. If a term rewriting system fulfills $t_1 < t_2$ according to the Knuth-Bendix order for all rules $t_1 \to t_2$ then the term rewriting system must be *terminating*. This means that there cannot exist infinite consecutive forwards rule applications for a term. For the term rewriting system shown above, there exists such an order, so it is a *terminating* rewriting system.

Similar to the critical pairs for HRGs introduced in Section 2.3.2, there are critical pairs for term rewriting systems. Note that for term rewriting systems we consider *forward* confluence, so the direction of rule application is switched here. For the term rewriting system from above, the following critical pair can be derived: $s(x) \leftarrow^{r1} s(x + 0) \to^{r2} x + s(0)$. It is not possible to apply any further rewrite steps to $s(x)$ and $x + s(0)$, therefore those terms are called *irreducible*. Both irreducible terms are not equivalent, which means that the critical pair is not joinable and the term rewriting system is not confluent.

The Knuth-Bendix completion algorithm [8] takes a term rewriting system and a matching Knuth-Bendix order as its input. Then the algorithm computes all critical pairs and for each critical pair, a new rewrite rule is added. The new rewrite rule contains the two irreducible terms of the critical pair as LHS and right-hand side (RHS). Whether a term is used as LHS or RHS is determined by the Knuth-Bendix order so that the modified term rewriting system is still terminating. In case the two terms t_1 and t_2 are not comparable in the order, i.e. if $t_1 < t_2$ and $t_1 > t_2$ both do not hold, then it is not possible to add a new rule. In this case, the completion algorithm fails and is not able to make the term rewriting system confluent. Otherwise, a rule is added for each critical pair. Adding all the rules can introduce additional critical pairs. Therefore

it is necessary to recompute the critical pairs and add new rules for the new critical pairs. This process is repeated until the term rewriting system has no more critical pairs. In this case, the completion algorithm is successful and the modified term rewriting system is confluent.

3.2 Applicability to hyperedge replacement grammars

An important difference between term rewriting systems and HRGs is the asymmetric rule format of HRGs. In an HRG it is not always possible to add a rule between the two fully abstracted graphs of a critical pair, as the LHS of a rule can only contain a single nonterminal edge. If by chance one of the fully abstracted graphs contains just the handle of a nonterminal this would be possible, but this is not always the case. In Section 5.1.2, we present a completion heuristic for HRGs, which adds rules in this way, if it is possible.

Another difference is that the rules of term rewriting systems induce a congruence relation for terms. If there is a rewrite path between two terms t_1 and t_2, these terms can be considered as *congruent* ($t_1 \approx t_2$). If there are two connected rewrite paths $A \to^* B$ and $A \to^* C$, for some terms A, B and C, then this means that $A \approx B$ and $A \approx C$. Because of the transitivity of the congruence relation, this means that B and C should also be congruent. This means adding a rule $B \to C$ would not change the induced congruence relation. All rules added by the Knuth-Bendix completion algorithm, do not change the congruence relation of the terms, as the terms of the added rule must have been congruent before.

An HRG is usually not used to represent a congruence between the two sides of a rule. In the case of Attestor, the rules specify abstractions and concretizations of heap configurations. So if there are two abstraction paths with the same origin, $A \Leftarrow^* B$ and $A \Leftarrow^* C$, for some hypergraphs A, B and C, then adding a new rule $B \Rightarrow^* C$ *could change* the language of the grammar. Consider the case that the graphs B and C are already fully abstracted and B represents the handle of some nonterminal so it can be a valid LHS in an HRG. By adding the rule $B \to C$, the graph C is no longer fully abstracted and can be further abstracted to B. In general, adding such a rule increases the language of the grammar. For Attestor this could lead to issues in the analysis of a program, as it is possible that during the abstraction important information is lost.

When designing a completion algorithm for HRGs, it is therefore important to address these issues that can arise.

3.3 The Attributed Graph Grammar System

AGG [14] is a tool developed at the TU Berlin. It uses grammars related to the general graph transformation systems we introduced in Section 2.3.1. However, in AGG hyperedges are not supported directly, but it is possible to model a hypergraph as a classical bipartite graph. For each hyperedge, a node is added and the tentacles connecting the hyperedge with the attached nodes are represented by normal edges. This means that every HRG can be represented as a grammar in AGG.

One of the features of AGG is that it allows computing the critical pairs of a grammar. The joinability of the critical pairs is not computed and therefore the tool is not able to determine if a grammar is confluent or not. For the attributed graph transformation systems of AGG confluence is not decidable, which is the reason why this step is left out. However, the critical pair computation is an important step in determining if a grammar is confluent. In Section 4.3.3, we compare the runtime of AGG with our confluence detection implementation.

We are not aware of any other implementation that computes critical pairs or decides confluence of graph grammars.

4 Confluence Detection

This chapter describes an efficient way to compute the critical pairs of an hyperedge replacement grammar (HRG) and how the joinability is computed to determine if the grammar is backward confluent. Our algorithm is based on the work by [5, page 45] that describe the general idea, how critical pairs can be computed for general graph transformation systems. We adapted these ideas to the framework of HRGs and provide a method to efficiently enumerate the critical pairs, with the aid of what we call overlappings. In the implementation section, we present insights into more specific details on how the algorithm was implemented into the Attestor tool. Lastly, we evaluate the algorithm to make conclusions about its performance.

4.1 Algorithm

The input to the algorithm is an HRG which fulfills the conditions of a data structure grammar as defined in Section 2.4.

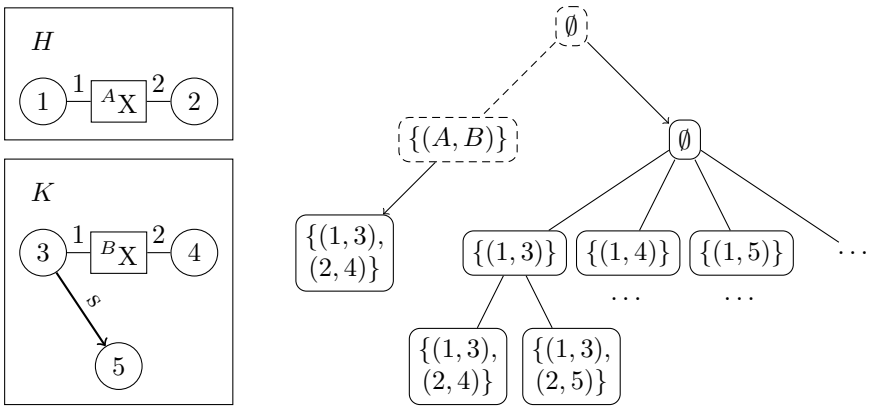
To determine if a grammar is confluent it is necessary to compute all possible critical pairs. In order to find a critial pair, the algorithm inspects certain graphs that contain two right-hand sides (RHS's) of the grammar. We call those graphs *joint graphs*. A joint graph can be created by *merging* the two RHS's of the rules were certain edges or nodes are mapped to a single edge or node in the joint graph. This merging of graphs can be understood as a relation between the nodes and edges of the two RHS's. We call those relations overlappings because they can be understood as drawing one graph ontop of the other were certain nodes and edges overlap each other. A node can either overlap with no node or exactly one node from the other RHS. The same holds for the edges of the RHS's.

Our algorithm works in three phases: In the *first* phase, all combinations of two rules in the grammar are enumerated. For those two rules, it is necessary to consider all possible graphs that can be created by *overlapping* the two RHS's. These possible overlappings are enumerated in the *second* phase. In the *third* phase, a joint graph is computed based on the current overlapping. It is then checked if the current configuration represents a critical pair. If it is a critical pair, then it is checked for strong joinability. If a critical pair is found that is not strongly joinable, then the HRG is *not confluent*. If all critical pairs are strongly joinable, the HRG *is confluent*.

In this section, we present a way to enumerate the overlappings required for the second phase and explain how to determine if a critical pair is strongly joinable. Lastly, we summarize the full algorithm by providing a simplified pseudocode implementation of the confluence detection algorithm.

4.1.1 Overlapping of Edges

The enumeration of the possible overlappings of two sets of edges, say E_1 and E_2 can be understood as the traversal of a tree. Each tree node represents a set of tuples from $E_1 \times E_2$. This set defines equivalences between the two RHS's and all nodes in the tree define all possible overlappings. The children of a node contain all tuples from the parent and one additional tuple where none of the two elements of the tuple is already contained in another tuple of this node. Note that this tree might contain the same overlapping multiple times, because the order in which tuples are selected does not matter. A solution to this issue is to define a total order on the tuples and requiring that the new tuple in each child has to be greater than all the other already present tuples. The resulting tree contains all different overlappings only once. It is possible to prune certain branches



(a) RHS's of Current Rules (b) Edge and Node Overlappings

Figure 4.1: Hierarchy of edge and node overlappings

in this tree if there are no overlappings that might lead to a valid critical pair. The way pruning is implemented is described in more detail in Section 4.2.4.

4.1.2 Overlapping of Nodes

Based on a specific edge overlapping all possible overlappings of nodes are computed. The given edge overlapping defines a set of node equivalences because for a specific edge equivalence all nodes connected to the i^{th} tentacle edge must be equivalent to the nodes connected to the i^{th} tentacle of the other edge. Note that for nonterminal edges all its connected nodes may be in the overlapping, but the nonterminal edges are not. For selector edges this does not hold because in case the selector is not in the overlapping it means that the joint graph will contain the selector twice. However, if the source nodes are equivalent, this would violate the condition that each node cannot have multiple outgoing selectors of the same type.

4.1.3 Complete Overlapping Tree

The way overlappings are enumerated is visualized in Figure 4.1. The two RHS's that the overlappings are based on is shown in Figure 4.1(a). Nodes are represented by circles with an id and nonterminal edges are represented by boxes. Each nonterminal edge has a unique id, which is shown in the top left corner. The edge overlappings are represented by dashed boxes and the boxes with straight lines represent node overlappings.

There are two edge overlappings: The empty overlapping and the overlapping where the nonterminal edges with the ids A and B are matched. There is no edge overlapping that contains the selector edge from K because there is no selector edge of this type in H.

Now we focus on the possible node overlappings that are based on the $\{(A, B)\}$ edge overlapping. It immediately leads to the node overlapping $\{(1, 3), (2, 4)\}$ because the nodes at the tentacles of the nonterminal edge have to match. There are no further node overlappings from here because there are no more free nodes in H, meaning all nodes are already in the overlapping.

Now we describe the node overlappings that depend directly on the empty edge overlapping. An empty edge overlapping always leads to an empty node overlapping. The children of the empty node overlapping are all possible tuples from $N_1 \times N_2$ where N_i is the set of nodes from RHS i. For simplicity, we now only look at the subtree of the node overlapping $\{(1, 3)\}$. In H only node 2 is still free and in K node 4 and node 5 are free. This means that *all* possible descendants are the node overlappings $\{(1, 3), (2, 4)\}$ and $\{(1, 3), (2, 5)\}$.

This tree is traversed for each combination of rules. Each node overlapping (except the empty node overlapping) defines a joint graph, which is a single component if both RHS's are single-

component graphs. The tree contains all possible joint graphs that can be computed from both RHS's, but not all those joint graphs lead to valid critical pairs. It is possible to prune complete subtrees, where no overlapping can lead to a new critical pair. This can increase the efficiency of the algorithm because the number of possible overlappings that grows exponentially with the height in the tree is reduced.

4.1.4 Critical Pair Computation

Every node overlapping and the corresponding edge overlapping define a joint graph were the nodes and edges are joined according to the given equivalences. To get the critical pair the two rules, used to create the joint graph, have to be applied in reverse. The mapping of the RHS's into the joint graph can be obtained easily because the joint graph was created from these RHS's. Note that other rules and mappings could also match the given joint graph, but it is only necessary to look at the specific mapping defined by the overlappings. The critical pair lemma [6] guarantees that the other matchings are considered in a different step.

The two rule applications might be independent in which case this joint graph is discarded, because it does not lead to a critical pair. Otherwise, the two rules are applied to the joint graph leading to two different graphs (G_1 and G_2). For these two graphs, a canonicalization step is executed, i. e., all rules of the grammar are applied backward as long as there is still a matching rule.

After the canonicalization, the two graphs are checked for isomorphism. In order for the grammar to be confluent the critical pair needs to be strongly joinable, which requires a special isomorphism. All nodes from the joint graph which have not been removed in both canonicalized graphs need to be equivalent between the two graphs in this isomorphism. If there are only isomorphisms without this condition then the critical pair is weakly joinable.

In our case, the given grammar is only confluent if all critical pairs are strongly joinable. If a critical pair is weakly joinable, the grammar is not confluent as explained in Section 2.3.2. We call critical pairs that are not strongly joinable violating critical pairs (VCPs).

4.1.5 Complete Algorithm Overview

The pseudocode in Algorithm 1 summarizes how the algorithm works as a whole.

First, it enumerates all pairs of rules of the grammar. If nodes in the rules can be joined, all possibilities of how the nodes can be joined, constitute different rules. Then the edge and node overlappings are enumerated in Line 5 and Line 6. For each of the overlappings, it is checked if they are strongly joinable or not. If the current overlapping creates a VCP, then the grammar is not confluent and the algorithm returns *false*. Otherwise, the algorithm continues by checking the other critical pairs. If no overlapping creates a VCP then the grammar is confluent and the algorithm returns *true*.

The joinability check in Line 8 is explained in more detail in Algorithm 2. In Line 2, the joint graph is computed from the current overlapping. When creating the joint graph it is important to keep track of the matchings $m1$ and $m2$, which determine how the respective RHS matches the joint graph. There might be different possibilities that an RHS can be matched in the joint graph, but the current overlapping defines one unique way. It is then necessary to check if the abstraction by both rules is independent, which means if it is possible to apply both rules one after the other with the given matchings. The abstractions are independent if the overlapping does not include any edges and only external nodes. Since external nodes are not removed during the abstraction, this means that no part of the graph gets removed that would be required to match the other RHS.

If the abstractions are independent, the current overlapping does not define a critical pair. In this case, the canonicalization can immediately abstract using the other rule which leads to the same graph for G_1' and G_2'. Hence, they can always be strongly joinable.

Note that the canonicalization is not deterministic if the grammar is not confluent. This might mean that the canonicalization step finds two fully abstracted graphs G_1' and G_2' which are strongly joinable by coincidence, but there must also exist another pair of valid canonicalized graphs for

Algorithm 1 Confluence detection algorithm

Input: *grammar* - An HRG
Output: A boolean indicating whether the input grammar is confluent or not.

```
 1: function IsConfluent(grammar)
 2:     for all (rule1, rule2) ∈ AllPairsOfRules(grammar) do        ▷ Rules can be equal
 3:         (nonterminal1, rhs1) ← rule1
 4:         (nonterminal2, rhs2) ← rule2
 5:         for all eOverlap ∈ ViableEdgeOverlaps(rhs1, rhs2) do
 6:             for all nOverlap ∈ ViableNodeOverlaps(rhs1, rhs2, eOverlap) do
 7:                 overlap ← (eOverlap, nOverlap)
 8:                 if ¬IsStronglyJoinable(rule1, rule2, overlap) then
 9:                     return false
10:                 end if
11:             end for
12:         end for
13:     end for
14:     return true
15: end function
```

which this would not hold. The critical pair lemma guarantees that there is another critical pair where this cannot occur. The reason for this is explained in more detail in Section 4.1.6.

In the case that two abstraction steps are not independent, both rules are applied to the joint graph and both resulting graphs are fully abstracted. In Line 10 the set of nodes is computed that were not removed from the joint graph during both abstraction paths. Then it is necessary to check if the fully abstracted graphs are strongly joinable. This is the case if there is an isomorphism between G_1' and G_2' such that all nodes in *commonNodes* map to the same node in the isomorphism. If there is such an isomorphism, then the current critical pair is strongly joinable and the function returns *true*.

4.1.6 Critical Pair Canonicalization without Confluece

An issue that is described in Section 4.1.5 is that we need to canonicalize graphs and check for strong joinability without knowing whether the grammar is confluent. At first, it seems that this might be an issue because if the grammar is not confluent, then canonicalization is not deterministic and might by coincidence return graphs that are strongly joinable. The example in Figure 4.2 shows an example of graph abstractions to demonstrate why this is not an issue. Here the joint graph is abstracted to G_1 and G_2 with one rule application. The canonicalization returns the graphs G_1' and G_2' which are strongly joinable. However, there is another possible fully abstracted graph H', which is an abstraction from G_1 that is not strongly joinable with G_2'. This means there is a graph H, which contains a critical pair so that there are two possible canonicalizations possible. Note that we only consider HRGs where the abstraction is terminating so rules that just map a nonterminal to another nonterminal are not allowed. Hence, each abstraction step removes some elements from the graph. This means that the joint graph of the critical pair that is contained in H must be smaller than the joint graph we started with. So the confluence detection algorithm at some point has to enumerate this smaller critical pair. At some point, there cannot exist a smaller critical pair and the canonicalization has to produce two fully abstracted graphs that are not strongly joinable.

4.2 Implementation

In this section, we provide implementation-specific details. It is targeted at the implementation of the confluence detection algorithm for Attestor in Java. When implementing the algorithm for

Algorithm 2 Joinability check for confluence detection algorithm

Input: $rule1$, $rule2$ - Two rules of an HRG consisting of left-hand side and right-hand side
$overlap$ - Specifies how edges and nodes of the two right-hand sides overlap.
Output: A boolean indicating whether the input grammar is confluent or not.

```
1:  function IsStronglyJoinable(rule1, rule2, overlap)
2:      (G, m1, m2) ← JointGraphAndMatchings(rule1, rule2, overlap)
3:      if AbstractionsAreIndependent(G, rule1, m1, rule2, m2) then
4:          return true                                          ▷ Not a critical pair
5:      else
6:          G_1 ← ApplyRule(G, rule1, m1)                        ▷ Found a critical pair
7:          G'_1 ← Canonicalize(G_1, grammar)
8:          G_2 ← ApplyRule(G, rule2, m2)
9:          G'_2 ← Canonicalize(G_2, grammar)
10:         commonNodes ← CommonNodes(G'_1, G'_2)
11:         if ∃iso ∈ Isomorphism(G'_1, G'_2) : ∀n ∈ commonNodes : iso(n) = n then
12:             return true                                      ▷ Strongly joinable critical pair
13:         else
14:             return false                                     ▷ VCP
15:         end if
16:     end if
17: end function
```

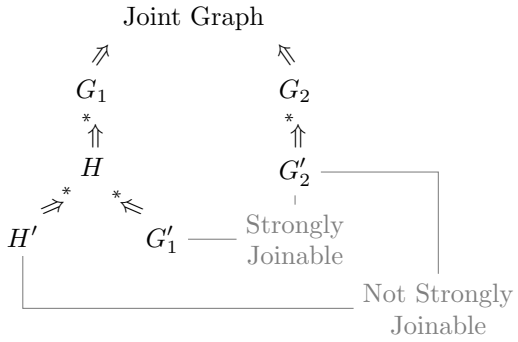


Figure 4.2: Nondeterminism in the canonicalization step of nonconfluent grammars

HRGs for other use cases it might be necessary to adapt it accordingly.

We first provide information about the format in which a grammar is stored in Attestor to better understand how the rules are accessed. Then we explain how the existing code of Attestor can be used to compute the joinability of critical pairs. We give insight into our implementation of overlappings and the used data structures and explain all possible ways that overlappings can be pruned in Attestor.

4.2.1 Grammar Format

A grammar is a set of rules and each rule consists of a nonterminal (left side of rule) and a heap configuration (right side of rule). Every nonterminal has a rank and a label with which it can be identified. A nonterminal in Attestor also stores which of its tentacles are reduction tentacles. A heap configuration is a graph with nodes, nonterminal edges and selector edges. For each node, it is possible to query which nonterminal edges and selectors are connected and for each nonterminal edge, it is possible to query the connected nodes. Nodes and nonterminal edges have integer ids, which are unique within a single heap configuration. Selector edges can only be identified by their source node and their label. In the following, we refer to nodes, selector edges, and nonterminal

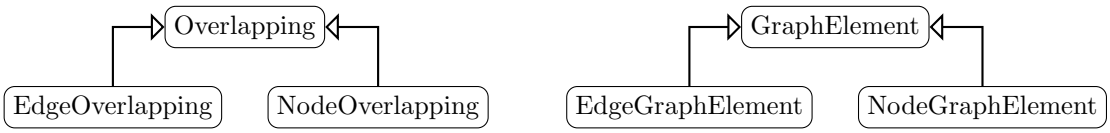


Figure 4.3: Overview of important classes for overlappings

edges as *graph elements*.

4.2.2 Detection of Strongly Joinable Graphs

When deciding if a critical pair is strongly joinable it is not enough to just check for an isomorphism. An isomorphism between the fully abstracted graphs only shows weak joinability. During the two abstraction paths starting from the joint graph, some nodes may get removed. The nodes that remain during both abstraction paths (called persistent nodes) have to match for an isomorphism to show strong joinability.

Attestor already provides a function that allows an isomorphism check. The external nodes of the two graphs have to match according to their external indices for an isomorphism. The nodes can be identified by ids that do not change during the abstraction steps. Therefore it is possible to compute the intersection of the ids of both fully abstracted heap configurations to obtain the set of persistent nodes. Note that this only works because when abstracting it is not possible that new nodes get created. Otherwise, this method could wrongfully detect newly created nodes as persistent nodes.

This means that it is possible to declare all persistent nodes external in the correct order and then Attestor only finds an isomorphism iff the critical pair is strongly joinable.

The correct alignment of persistent nodes in the isomorphism also implies correct alignment for persistent edges that might occur. In case an edge is persistent, all connected nodes must also be persistent nodes. In this case, the correct alignment of the connected nodes also implies the correct alignment of the edge.

4.2.3 Implementation Design of Overlapping Enumeration

The enumeration of overlappings is of utmost importance for the efficiency of the confluence detection algorithm because the number of all possible overlappings grows exponentially with the graph size. In this section, we give insights into design choices we made for the implementation of the overlapping class.

As the enumeration of overlappings is the most complex part of the confluence detection, the responsible classes are covered by JUnit tests. Most notably we constructed several test cases that ensure the correct expected order of overlappings and ensure that no overlapping is missed.

Overview of Important Classes for Overlappings

Every overlapping is represented by the `Overlapping` class. An object of this class defines how two sets of `GraphElement`s overlap. A `GraphElement` can be a node, selector edge or hyperedge. This relationship is illustrated in Figure 4.3.

The `Overlapping` class is a generic abstract class that contains the logic to compute the next `GraphElement` object and keeps track of the graph elements in the overlapping and the remaining graph elements. The `EdgeOverlapping` class fixes the generic parameter of the `Overlapping` class to the `EdgeGraphElement` class so it is only possible to specify overlappings of edges. Similarly, the `NodeOverlapping` class fixes it to compute overlappings of nodes only.

The overlapping subclasses also implement a method `isNextPairCompatible(newEquivalence)` that is called by the `Overlapping` class to check if an overlapping with the additional new equivalence would be valid. Note that this reduces the number of object creations because no new overlapping

object is created when it would not be valid in comparison to the alternative of just creating all possible child overlappings and then deleting invalid overlappings.

The `Overlapping` class implements the `Iterable<Overlapping>` interface to iterate over all overlappings, which are descendants of the current overlapping in the overlapping tree.

Usage of Data Structures

The data structures that are used in the overlapping are very important for the speed of the confluence detection. The confluence detection algorithm is called many times during the grammar completion algorithm, so the fast execution of the confluence detection is very important.

Each overlapping object contains two tree sets (for each RHS) that store the remaining graph elements that are not yet in the overlapping and might be included in following overlappings. A tree set was chosen because it allows fast queries for the lowest element in the set and the successor element according to the element order, which is used to obtain the next overlappings.

The object also contains two hash maps which map the elements of one RHS to the other for all graph elements that are in the overlapping. The second hash map is always the inverted mapping which could be omitted but is still included to simplify the implementation.

An overlapping object also contains the last graph element equivalence that was added to the overlapping. For the empty overlapping (no graph element in the intersection) the value is null. For all other overlappings, the last graph element equivalence must be greater than all other equivalences in the overlapping according to the defined order.

When creating a new overlapping object all those data structures are copied and then modified to include the new equivalence. It might be possible to optimize memory usage and execution time by storing just the new equivalence and to compute the necessary information only when required. This approach would, however, add a lot of complexity and we do not expect this to significantly improve performance.

4.2.4 Overlapping Pruning

In this section, we present all different mechanisms that we implemented to prune the tree of possible overlappings. If an overlapping is pruned, all descendant overlappings in the overlapping tree will not be considered so only overlappings where this does not matter must be pruned. Pruning is necessary, because of the exponential growth of the overlapping tree, so pruning more overlappings leads to faster execution of the confluence detection algorithm.

Type Compatibility

In Attestor not only the edges have certain types, but also the nodes. For an overlapping, the types between two elements of an equivalency have to match.

Incompatible Induced Node Equivalences

Each edge equivalency also implies several node equivalencies. It might be the case that multiple edge overlappings induce conflicting node equivalences because a node in one RHS cannot map to multiple nodes in another RHS. For this reason, each edge overlapping also contains two hash maps that specify the node equivalences. These hash maps simplify checking whether, for all connected nodes of a new edge equivalency, there is any violation. It also simplifies the construction of a node overlapping from the edge overlapping.

Violation Points in Node Overlappings

When an abstraction is applied to a heap configuration then all internal nodes (nodes that get removed during abstraction) must not be connected to any other edges that would not be removed. Such a node is called a violation point and all node overlappings that contain a violation point are

pruned. This case is also referred to as *dangling edge condition* because the remaining edge would be missing a node to connect to.

The edge that contributes to this violation is not deleted in any of the descendant overlappings, so the violation point will be part of all descending overlappings. Therefore, it is possible to prune all overlappings with a violation point.

Avoiding Conflicting Outgoing Selectors

For heap configurations, it is not allowed that a node has multiple outgoing selectors of the same type. For every new node equivalence in a node overlapping the two sets of outgoing selector labels of both RHS's are computed. If those sets of labels are disjoint, there is no issue. Otherwise this node overlapping is pruned.

Note that this also removes the node overlapping where the target nodes are equivalent in the overlapping. It is not an issue that this case is removed because it is considered in the node overlappings that are based on an edge overlapping where both of these selector edges are equivalent.

Ensuring `NULL` Equivalence

In Attestor, nodes represent objects in memory. In some grammars, it is known that certain nodes are `NULL`. There should only be one `NULL` node in a graph so it does not make sense to compute joint graphs with multiple `NULL` nodes. Therefore, it is possible to prune all overlappings that don't allow the `NULL` nodes of the two RHS's to be joined.

Avoid Enumerating Same Overlappings

As described previously we need to define an order on the graph elements to avoid enumerating the same overlapping twice. All overlappings are pruned where the new equivalence is considered to be lower than the already present equivalences.

4.2.5 Order on Graph Element Equivalences

Each overlapping represents a set of graph element equivalences. A graph element equivalence specifies what element from one RHS is equivalent to another element from the other RHS. We must define an order on the graph element equivalences to systematically enumerate all possible overlappings. This order is a pointwise extension of an order on the graph elements themselves.

The order on the graph elements is based on its id in the heap configuration. For selector edges, the id of the source node is used primarily to determine the order and if the id of the source node is equal, then Java's `String.compareTo()` function is used.

The enumeration of edges and nodes is done separately so there is no need to be able to compare selector edges and nodes. This means that the case that the ids are equal, but only one graph element is a selector edge, does not occur.

4.2.6 LaTeX Debug Output

To easily verify that the critical pairs found by the implementation are indeed correct, it is possible to output all critical pairs into a LaTeX file. This file can be compiled using the lualatex compiler and the resulting pdf file shows for each critical pair the joint graph, which two rules are applied, the two graphs after one abstraction step and the respective canonicalized graphs. It is also possible to generate a pdf with all rules of an HRG which helps to verify that the creation of the joint graph is correct.

The graph layout is computed using a force-directed layout algorithm [15] that is included in the TikZ LaTeX package. The Java code generates the LaTeX file by first pasting a fixed preamble containing multiple macros. In the document, the Java code inserts all information of the critical pair and the contained graphs by setting PGF values. PGF is another latex package which is also used by TikZ that among other things allows to set and read key-value pairs. For each critical pair,

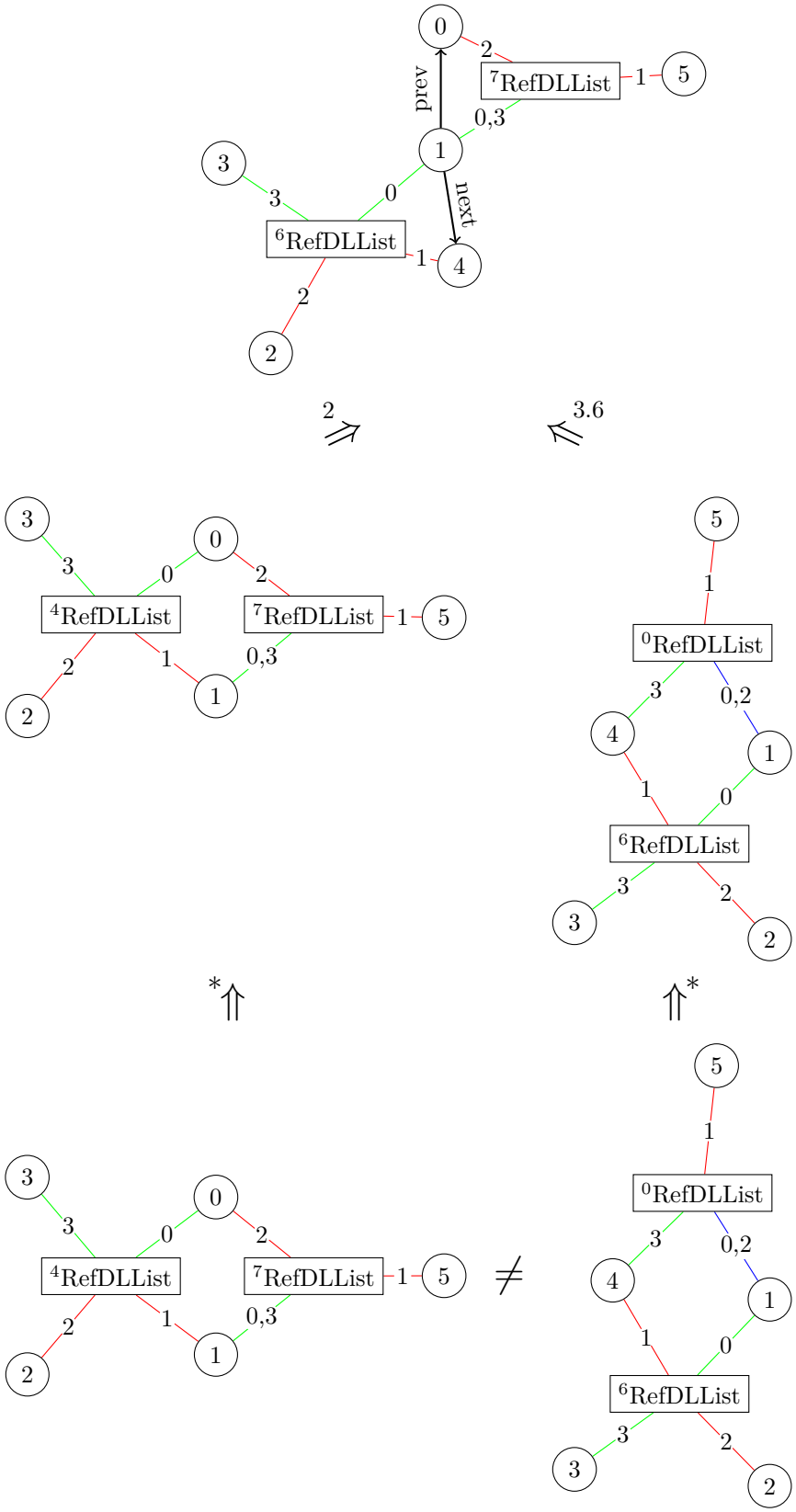


Figure 4.4: Example of automatic generated critical pair of DLList grammar

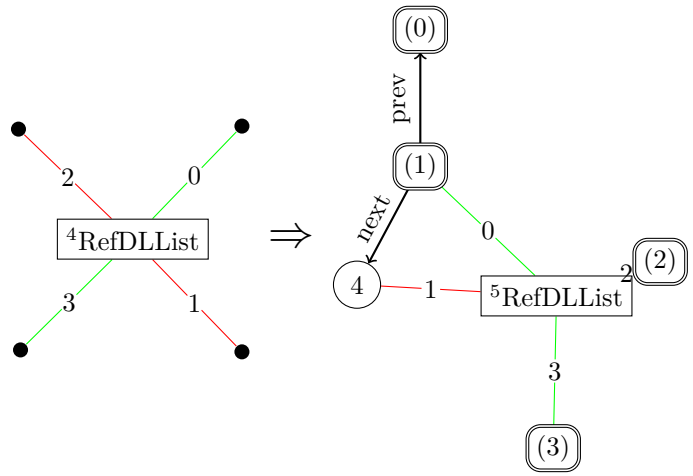


Figure 4.5: Automatically generated layout of rule 2 of DLList grammar

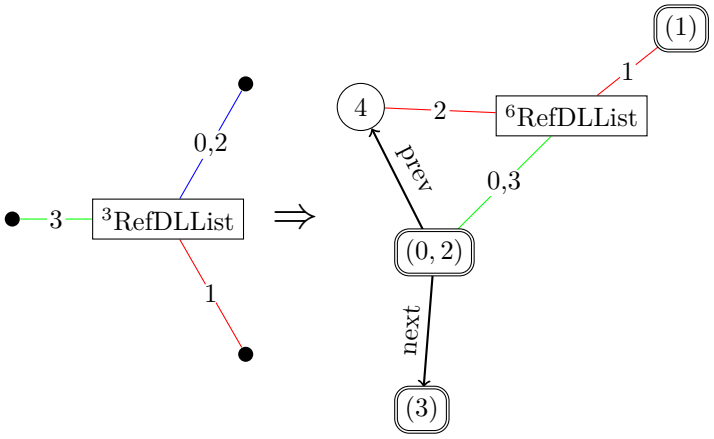


Figure 4.6: Automatically generated layout of rule 3.6 of DLList grammar

a macro is called which reads the PGF keys that were set previously to display the information about the current critical pair.

In Figure 4.4 an excerpt from the automatically generated critical pair analysis report is shown. It shows the joint graph, the two immediate rule applications, and the two fully abstracted graphs. Numbers at the nodes and nonterminal edges visualize how nodes and nonterminal match with the joint graph. The shown critical pair is not joinable because there is no isomorphism between the fully abstracted graphs. Even though the graphs look similar the left graph contains a connection to a node with tentacle $(0, 3)$, but there is no such connection in the other graph. At the abstraction arrows at the top of the figure, the identifiers (2 and 3.6) of the rules of the grammar are shown.

These two rules are shown in Figure 4.5 and Figure 4.6. Here the nodes with a double border show external nodes. When applying a rule the nodes connected to a tentacle with certain indices have to match the node with the corresponding indices shown in the double border node. All those rules are also generated automatically to easily check that the computed critical pairs are correct.

Using these generated pdfs we were able to verify with some of the computed critical pairs that the critical pair computation works as expected.

4.3 Evaluation

The confluence detection implementation was evaluated on a set of different HRGs. Attestor already includes some predefined grammars that are all used for the benchmarks. These grammars represent the following common data structures: A singly-linked list (SLList), doubly-linked list (DLList) and a binary tree (BT) grammar. The abbreviation in brackets is used in the following figures to refer to these grammars.

We also evaluated the algorithm with grammars from other work. We tested a grammar for trees with linked leaves (LinkedTree2) from [9] and grammars for in-trees (InTree, InTreeLinkedLeaves) from [7]. In-trees are a special tree data structure where children have pointers to their parent instead of the other way round.

We also created another grammar for trees with linked leaves (LinkedTree1), which has fewer external nodes.

The grammar for doubly-linked lists in Attestor is quite complex because it was created to allow concretization into any direction in the list and it creates pointers to a null node for the first and last element. For this reason, we also created a simpler grammar for doubly-linked lists (SimpleDLL), which contains fewer rules with smaller graphs.

In this section, we first explain how we evaluated our implementation and then present the results. Finally, we compare our implementation with the AGG tool [14], which can compute critical pairs for general graph transformation systems.

4.3.1 Methodology

The evaluation is done using a benchmark main method. The benchmark method executes confluence detection for a set of certain HRGs. The method does not execute a confluence check twice for the same grammar to avoid distorted results due to javas caching optimizations. To get multiple timing results to compute the average runtime, the java code is rerun multiple times. This ensures that the Java virtual machine (JVM) is restarted and that the cache is discarded.

A java benchmark framework was not used because the benchmarks here require more flexible time measurement. We use multiple separate stopwatch objects that are started and stopped at specific locations in the code to measure not only the whole runtime but for example how much time is used for computing the edge overlapping alone. These stopwatch objects use Javas `ThreadMXBean.getCurrentThreadCpuTime()` method to get a timestamp to compute the elapsed time between starting and stopping the stopwatch. The method `getCurrentThreadCpuTime()` was used instead of a real-time timestamp to only measure the time that specific java thread is running. To mitigate any issues that might arise due to context switches we do not run more

threads than cores are available on the computer running the benchmark. Note that this way of measuring the runtime does not include the JVM startup time.

During all tests, the memory consumption has never been an issue and it is possible to run all benchmarks with less than 500 MB. For this reason, we did not further evaluate memory consumption, because it is not a limiting factor.

4.3.2 Results

To evaluate our implementation, we first provide runtime measurements for the different grammars. Additionally, we also count how many overlappings are pruned during the execution. This should justify the necessity of our pruning implementation. The pruning statistic is evaluated and discussed for the DLList grammar, but we uploaded additional statistics for the other grammars[1].

Runtime

Figure 4.7 shows an overview of the runtimes for the different grammars. The runtime is split up into four categories. In the edge overlapping category, it is measured how much time is spent on trying to find valid edge overlappings for which there are possible node overlappings. Similar in the node overlapping category the time it takes to find valid node overlappings is measured. The joinability check phase measures the time used for computing the critical pair, executing the two canonicalizations and determining the joinability. The category *other* shows any other time for example used to enumerate the different rule combinations.

The chart shows that the runtime is below 0.5 s for all grammars. The time required to compute the overlappings in relation to the time spent on determining the joinability varies for the different grammars but is roughly equal.

Table 4.1 shows different properties of the used grammars. It shows the number of rules which includes the collapsed rules that are obtained from the original rules. It also shows the maximum number of external nodes, all nodes and edges (selectors and nonterminals) that occur in an RHS of the grammar. We see that the runtime correlates with the number of rules, which makes sense because the algorithm has to check all combinations of two rules. One outlier, however, is the BT grammar. This grammar has the second-highest number of rules but still completes in under 0.04 s. A reason for this might be that the BT grammar is the only grammar that uses `NULL` nodes, which allow more overlappings to be pruned.

Grammar	Rules	Max Ext. Nodes	Max Nodes	Max Edges	Runtime
SimpleDLL	2	2	3	3	0.8 ms
SLList	6	2	3	2	3 ms
LinkedTree1	10	3	7	5	25 ms
BT	23	3	5	4	30 ms
InTreeLinkedLeaves	15	3	4	2	87 ms
InTree	12	2	5	4	107 ms
LinkedTree2	20	4	7	7	330 ms
DLList	40	4	6	4	457 ms

Table 4.1: Characteristic properties of the different grammars

Pruning Statistic

To get an idea of how many states are traversed and checked we provide the chart in Figure 4.8. The level here corresponds to the height in the tree from Figure 4.1(b). It shows the number of edge overlappings and node overlappings which are pruned at the respective level. Because

[1] https://github.com/Johannes-S/confluence — Hash: 3149498bea27cf3cbb5b4c6682383b6b2d519faa

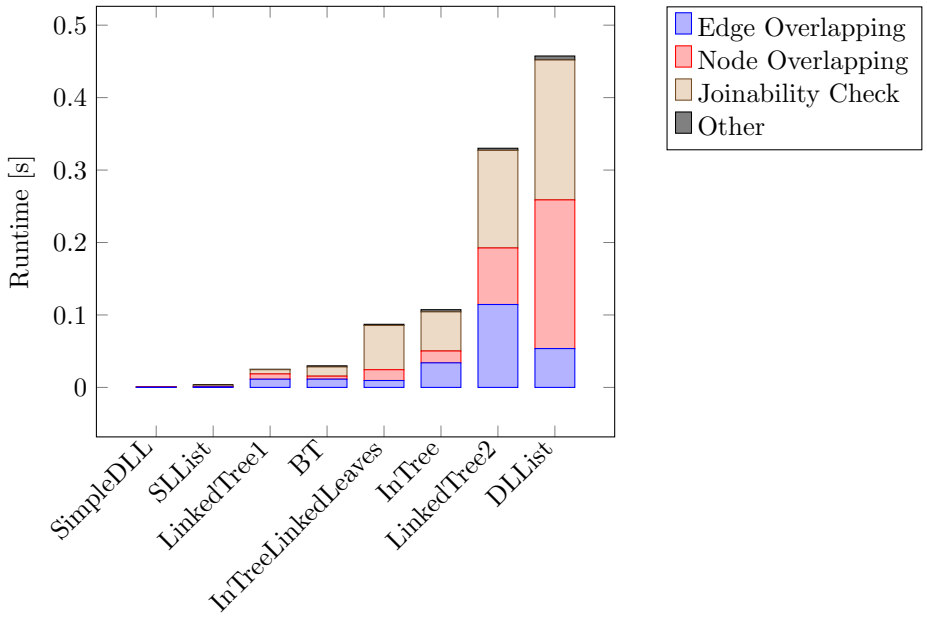


Figure 4.7: Confluence detection runtime of different grammars

the number of all *possible* overlappings grows exponentially with the level it is beneficial if the overlapping pruning step can prune overlappings as early as possible.

The chart shows that the edge overlapping pruning can prune many overlappings at level 0 and level 1. At level 4 there are no more possible edge overlappings because in the DLList grammar there are at most 4 edges in an RHS so it is not possible to prune any more edge overlappings.

The node overlappings cannot start at level 0 because they follow after an edge overlapping. At level 1 to level 5 the number of pruned node overlappings is in the 10^4 to 10^5 magnitude.

It is difficult to quantitatively measure the quality of the pruning, but this result indicates that the implemented pruning is successful. For comparison, in a bad pruning implementation, we would expect the number of pruned overlappings to grow exponentially with the pruning level and the ratio of pruned node overlappings at higher levels would be much higher.

The results show that the number of overlappings that are pruned is *not* rising exponentially. The number of pruned node overlappings is higher than the number of valid overlappings with a factor of 10^3, which still seems reasonable.

It is difficult to quantify the effectiveness of the pruning, but these results indicate that the pruning is working.

4.3.3 Comparison with AGG

AGG [14] is a tool developed at the TU Berlin for attributed graph transformation systems. It is written in Java and provides a feature to compute the critical pairs for general graph transformation systems. The joinability of the critical pairs is not determined so the tool is not able to check confluence.

The grammars accepted by AGG are general graph transformation systems and not HRGs. It is possible to translate the HRGs from Attestor into a format that can be processed by AGG. AGG also does not work on hypergraphs, but normal graphs with edges of rank two. Nonterminal edges are therefore modeled by a node of a distinct type in AGG and the tentacles are represented by normal edges that connect the nonterminal edge to the connected nodes. The tentacle edges are assigned types according to the tentacle index.

Collapsed rules are also not directly supported by AGG, which means it is not possible that one

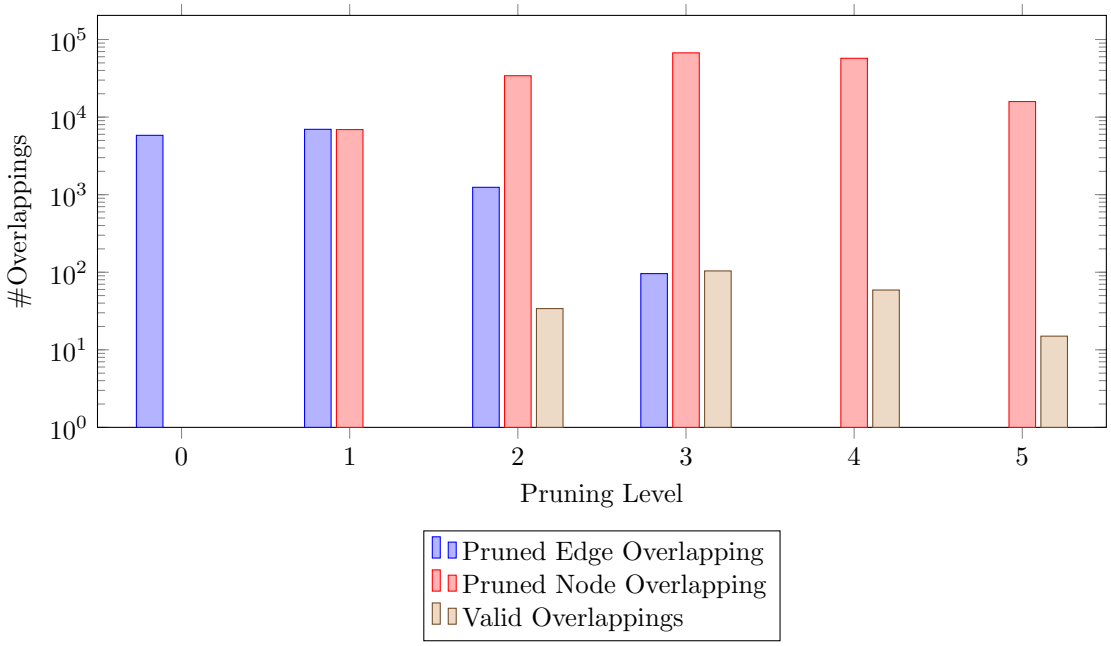


Figure 4.8: Overlapping pruning statistic of DLList grammar

node in a graph is matched to multiple nodes in a rule. In Attestor collapsed rules, with joined nodes of the rules, are created at the start. These collapsed rules can be imported into AGG to support the same number of rules.

To compare our algorithm with this tool, we have implemented an exporter that converts HRGs from Attestor to a special XML format required by the AGG tool. The critical pair computation of AGG does not include a joinability check and just computes the critical pairs, as this is a large part of our confluence detection it is still suitable for comparison.

Grammar	AGG	Attestor Critical Pairs	Attestor Complete
SimpleDLL	264 ms	1 ms	1 ms
SLList	242 ms	1 ms	3 ms
LinkedTree1	>24 h	19 ms	25 ms
BT	87 697 ms	17 ms	30 ms
InTreeLinkedLeaves	24 936 ms	26 ms	87 ms
InTree	0.8 h	53 ms	107 ms
LinkedTree2	>24 h	195 ms	330 ms
DLList	18.6 h	264 ms	457 ms

Table 4.2: Runtime comparison with AGG on different grammars

In Table 4.2 the runtime of the AGG tool is compared to the confluence check of our implementation. The runtimes of our implementation for Attestor are shown for just the critical pair enumeration and also for the complete confluence detection that includes the joinability check. For smaller grammars like the SimpleDLL and SLList grammar, our tool is faster by a factor of 10^2, while for larger grammars like the DLList grammar it is faster by a factor of 10^5.

Note that the AGG tool can compute critical pairs for a larger class of graph transformation systems, which our implementation does not support. One reason for this large difference in speed might be because AGG does not support hyperedges. A single hyperedge translates to one node and multiple edges in AGG, which largely increases the number of possible overlappings. It could

be possible to further improve the runtime performance of AGG for the given grammar by changing the way hyperedges are encoded. For each hyperedge of rank i, we introduce $i + 1$ elements into the graph. One node that represents the hyperedge and i edges that represent the tentacles.

Our implementation also benefits from the fact that hyperedges are overlapped in the first step because the overlapping of two hyperedges already defines the overlapping of all connected nodes, which reduces the number of overlappings that have to be considered. In the grammars we converted for AGG there are no distinguishing features between normal nodes and nodes that represent hyperedges so it cannot apply such a technique.

5 Grammar Completion

Grammar completion is a procedure, which modifies a grammar to make it confluent. For some non-confluent grammars there exists a confluent grammar, where the languages of the two grammars are equivalent. Note that such a grammar need not exist in every case. However, if language equivalence is not required, it is always possible to find a confluent grammar with a larger language. A grammar with a nonterminal, whoose language contains every possible hypergraph is always confluent, as all backward rule applications eventually lead to a graph containing only this nonterminal. A completion algorithm that is not required to keep the languages equivalent could therefore always return such a grammar, which would not be very useful in practice. Ideally the completion algorithm should return a grammar where the extension of the language is minimal.

In this chapter, we present a completion algorithm that uses multiple heuristics, which all introduce small changes to the grammar. The confluence detection algorithm, presented in the last chapter, is used to determine the number of VCP of the modified grammar. A grammar with no violating critical pairs (VCPs) is confluent, so by only keeping changes that lower the number of VCPs, the grammar gets closer to beeing confluent in each step.

Our approach does not guarantee to find a grammar whoose language is closest to the language of the original grammar, but in this chapter we discuss some of the precautions we took to prevent the return of a trivial grammar.

5.1 Algorithm

The completion algorithm is split into three different parts: A collection of completion heuristics, grammar validity checks, and a completion strategy. The completion strategy uses the completion heuristics to modify the grammar to reduce the number VCPs and it uses the grammar validity checks to determine if the modified grammar satisfies other requirements, e.g. that the grammar is a data structe grammar or that it is locally confluent. In the following sections, these three parts are explained in more detail.

5.1.1 Greedy Completion Strategy

The purpose of the completion strategy is to find a combination of possible heuristic applications such that the resulting grammar is confluent and fulfills the grammar validity checks. For simplicity, we decided to implement the completion strategy as a greedy algorithm. It works by applying a heuristic on the current grammar and then using the confluence detection algorithm described in Chapter 4 to determine the number of VCPs. The greedy algorithm chooses the first grammar created by a heuristic with a lower number of VCPs than the current grammar. In the following, we will present this algorithm in more detail.

In Algorithm 3, the pseudocode for the algorithm is shown. The *greedy completion* function uses the boolean variable *madeProgress* to detect when no heuristic provides any improvement to the number of VCPs. This variable is set to *true* initially to enter the loop in Line 3. In the loop, it is set to $false$ so the loop is exited if no progress has been made.

Then the completion strategy loops over all possible heuristics. The heuristic is applied to the grammar in Line 6. The result of the `ApplyHeuristic` function is a modified grammar for which the number of VCPs is computed, whose exact process is explained later in this section. If the number of VCPs of the grammar is zero then the grammar is confluent. In this case, the result is returned immediately in Line 10. In case the new grammar has fewer VCPs then before, the *grammar* variable is updated accordingly as seen in Line 12. As the number of VCPs has

Algorithm 3 Greedy completion strategy

Input: *grammar* - An HRG
Output: A confluent HRG, if possible by using the heuristics.

```
1: function GreedyCompletion(grammar)
2:     madeProgress ← true
3:     while madeProgress do
4:         madeProgress ← false
5:         for all currentHeuristic ∈ CompletionHeuristics do
6:             newGrammar ← ApplyHeuristic(currentHeuristic, grammar)
7:             oldCP ← NumberVCPs(grammar)
8:             newCP ← NumberVCPs(newGrammar)
9:             if newCP = 0 then
10:                return newGrammar                    ▷ Grammar is confluent
11:            else if newCP < oldCP then
12:                grammar ← newGrammar
13:                madeProgress ← true
14:            end if
15:        end for
16:    end while
17:    return newGrammar                    ▷ Failed to make grammar confluent
18: end function
```

been reduced successfully, the *madeProgress* variable is set to *true*. This will ensure that if the grammar is not confluent after trying all heuristics once, all heuristics are then checked again in case new VCPs can be removed by a heuristic that has been used before the VCP was created.

The pseudocode for the ApplyHeuristic function is shown in Algorithm 4. It applies the same heuristic multiple times consecutively to the grammar as long as the number of VCPs can still be reduced. Similar to the *madeProgress* variable before, a variable called *appliedHeuristic* tracks if the heuristic can still be applied and allows to abort if this is no longer possible.

In the loop in Line 5 the `GetNewGrammars` function is called. This function returns all possible modified grammars that can be created from the given grammar by applying the heuristic once. There are multiple possible grammars because the heuristic can target different VCPs or might provide different ways to remove a VCP. The details here depend on the heuristic that is currently used. For the new grammar, the number of VCPs is computed and if it is lower than the number VCPs before and the modified grammar satisfies the grammar validity checks, then the grammar is updated accordingly and the *appliedHeuristic* boolean is set to *true*. This indicates that it is necessary to check if the heuristic can be applied again. In Line 11, the loop iterating over the possible grammars is broken to trigger a recomputation of the possible new grammars. The recomputation is necessary to include the updated grammar in the result. The changed grammar might also introduce new VCPs which should be removed in further applications of the heuristic. If at some point no new grammar can further reduce the number of VCPs the outer loop is exited and the current grammar is returned.

The completion strategy described here is build so that the same heuristic is executed as long as it can decrease the number of VCPs. Only if there are no further improvements possible, does the algorithm move to the next heuristic. This allows setting a priority for the different heuristics by their order. This way the preferred heuristic is tried first and only in case no further improvements are possible are the other heuristics used.

The termination of the completion algorithm is guaranteed because there is only a finite number of heuristics and each heuristic can only generate a finite amount of new grammars. The number of VCPs is decreasing with every iteration from the outer loop of Algorithm 3 or Algorithm 4 or the loop is exited. This is the case because changes of heuristics are only applied if the number VCPs is strictly smaller than before.

Not every grammar can be made confluent using the given heuristics so it is still possible that

Algorithm 4 Application of completion heuristic

Input: *heuristic* - A completion heuristic
grammar - An HRG
Output: The modified HRG, if the heuristic can improve it. Otherwise, the original HRG.

```
1: function ApplyHeuristic(heuristic, grammar)
2:     appliedHeuristic ← true
3:     while appliedHeuristic do
4:         appliedHeuristic ← false
5:         for all newGrammar ∈ GetNewGrammars(heuristic, grammar) do
6:             oldCP ← NumberCriticalPairs(grammar)
7:             newCP ← NumberCriticalPairs(newGrammar)
8:             if newCP < oldCP ∧ GrammarIsValid(newGrammar) then
9:                 grammar ← newGrammar
10:                appliedHeuristic ← true
11:                break
12:            end if
13:        end for
14:    end while
15:    return grammar
16: end function
```

there are some remaining VCPs at the end. The algorithm still returns the confluence optimized grammar so the user can look at the remaining issues and can try to modify the grammar by hand to achieve confluence. There are different reasons, why the completion algorithm cannot make a grammar confluent. We allow the user to select which heuristics should be used and it could be the case that the selected heuristics alone are not enough to make the grammar confluent. Another reason is the greedy completion procedure that only applies a heuristics if it lowers the number VCPs. It is possible that at some point all heuristics lead to new critical pairs, which increase the number VCPs, but the newly introduced critical pairs could be resolved later. The greedy algorithm is not able to find such solutions. And lastly, the grammar validity checks might prevent that a confluent grammar is found. Recall that the grammar validity checks are used to impose additional requirements on the grammar, which could be violated by all modified grammars.

5.1.2 Completion Heuristics

In this section, we give more details about the different heuristics that are used by the completion strategy. Each heuristic is intended to change the grammar as little as possible. One trivial but useless completion approach would be to add a rule for all fully abstracted graphs of VCPs which maps a single new nonterminal to all those fully abstracted graphs. This approach would be useless in practice because when abstracting a graph a lot more information is lost in comparison to the original grammar. By modifying the grammar as little as possible, we hope to find a grammar that more closely resembles the language of the original grammar.

The following sections introduce the different heuristics that can be used to make an hyperedge replacement grammar (HRG) confluent.

Rule Restriction Heuristic

The rule restriction heuristic modifies the grammar by removing a single rule. It is only used to remove collapsed rules because there is a chance that some collapsed rules that were generated would have never been used in the analysis anyway. Removing original rules which are not collapsed rules is not advisable because they change the grammar fundamentally. However, it might still be useful to allow the heuristic to remove original rules to see which rules are problematic in a non-confluent grammar so the user can then modify the grammar per hand.

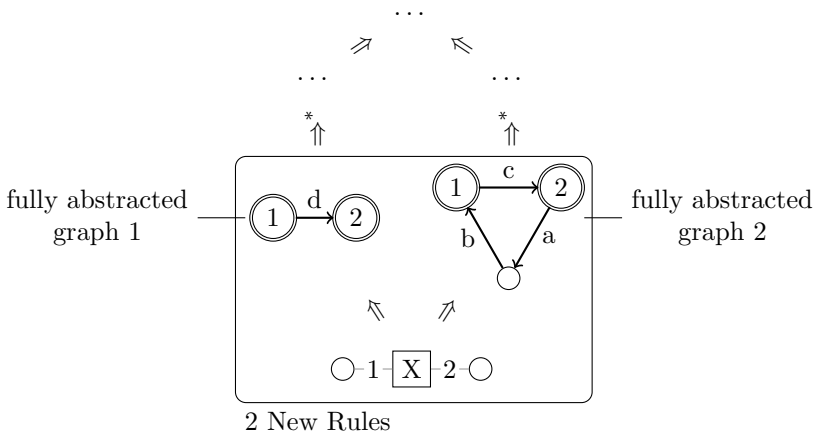


Figure 5.1: New nonterminal heuristic example

Removing a rule removes all VCPs that originate from this rule, but it could also create new VCPs because the canonicalization might return a different graph for another VCP. This means that a critical pair that was joinable before is not joinable anymore.

New Nonterminal Heuristic

This heuristic creates a new nonterminal for a VCP. Then two rules are added that map this new nonterminal to both fully abstracted graphs of the VCP. By adding these rules the previously fully abstracted graphs can be abstracted one step further to the new nonterminal and making the critical pair strongly joinable.

An example of this heuristic is shown in Figure 5.1. It shows the abstraction path of a critical pair. The two fully abstracted graphs are not isomorph and no rule in the *unmodified* grammar can be applied to the graphs. The heuristic modifies the grammar by adding two rules with the nonterminal label X as left-hand side (LHS), where X is a new label not used in the unmodified grammar. The right-hand sides (RHS's) of the rules are the two fully abstracted graphs, but some nodes of the graphs must be declared external so they are valid RHS's. It is arbitrary, which of the nodes are selected as external nodes. Adding the two rules means that the two graphs, which were fully abstracted in the unmodified grammar are no longer fully abstracted in the modified grammar. In the modified grammar both graphs can be further abstracted to the handle of X.

The rank of the nonterminal and the node can be arbitrary, but it seems reasonable to just use a rank of 1 and select a random node of each fully abstracted graph as the external node for the new rules.

If the rank is greater than 1, then the order of the external nodes is of importance. Recall that for strong joinability, the common nodes of the fully abstracted graphs must match in the isomorphism. This means that if some of the nodes of the fully abstracted graphs that are chosen as external nodes are common nodes, then every pair of common nodes must have the same index in the order of external nodes. Otherwise, the critical pair would only be weakly joinable and not strongly joinable.

Applying this heuristic for a VCP removes this VCP, but the newly added rules might introduce new VCPs.

Joining Nonterminals Heuristic

If two nonterminals have been created using the *new nonterminal heuristic*, those nonterminals can be joined so that all occurrences of both nonterminals are replaced by the joined nonterminal. This is achieved, by renaming all occurrences of one of the nonterminals in the grammar to the other nonterminal. The heuristic can only be done if the rank of the two nonterminals matches.

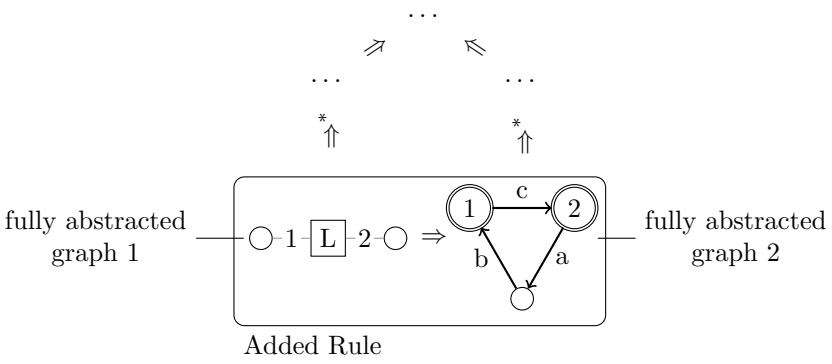


Figure 5.2: Single nonterminal heuristic example

This heuristic could remove VCPs that were introduced by applying the *new nonterminal heuristic* twice and creating a conflict between two RHS's of the new rules. A drawback is that this heuristic could modify the language too much. Using this heuristic in conjunction with the *new nonterminal heuristic*, it would be possible that all VCPs of a grammar are eliminated by having a single nonterminal that maps to all possible fully abstracted graphs of the VCPs. The resulting grammar would be confluent, but might not be very useful because when abstracting a graph a lot of information might get lost.

Single Nonterminal Heuristic

In the special case that one of the fully abstracted graphs of a VCP is just a single nonterminal it is possible to add a rule that maps this nonterminal to the other fully abstracted graph of the VCP.

Here the rank of the nonterminal is predefined. It would be possible to just select one node in the RHS and make it the only external node, meaning when abstracting using the added rule, all tentacles connect to this node. This would mean there is a collapsed rule without a corresponding original rule, which would still be possible. Alternatively, it is possible to just select several nodes of the RHS and make them external nodes in some order.

An example of this heuristic is shown in Figure 5.2. It shows the abstraction paths of a critical pair, which lead to the two non-joinable fully abstracted graphs with the unmodified grammar. The heuristic can be applied, as the fully abstracted graph on the left is a single nonterminal with label L. The grammar is modified, by adding a rule with L as LHS and the right fully abstracted graph as RHS. This means that the graph on the right can be further abstracted to the graph on the left, leading to a strongly joinable critical pair.

This heuristic is similar to the Knuth-Bendix completion algorithm presented in Section 3.1, as it just adds a single new rule containing the two conflicting elements of the critical pair.

Handle with Common Subgraph Heuristic

The case that one fully abstracted graph is just a single nonterminal with connected nodes does not occur very often. It is more probable that one fully abstracted graph contains a nonterminal and some subgraph that is also part of the other fully abstracted graph. Consider a VCP with the two fully abstracted graphs A_1, A_2, and the common subgraph S. If there is a nonterminal N and a graph G so that $A_1 = N \cup S$ and $A_2 = G \cup S$, then it is possible to achieve confluence by adding the new rule $N \Rightarrow G$. The example Figure 5.3 shows two fully abstracted graphs and the resulting rule as an example. Here the common subgraph S contains just the edge with the label s and the connected nodes. When abstracting graph A_1 with the new rule the resulting graph is isomorph A_2. The fully abstracted graphs of the VCPs are in this case strongly joinable and the heuristic could be applied successfully.

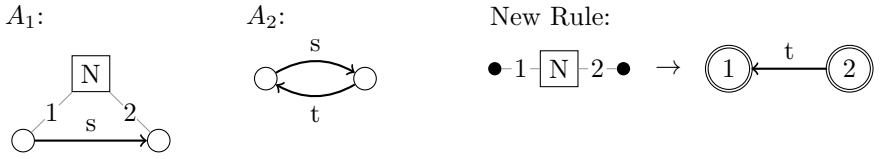


Figure 5.3: Example for handle with common subgraph heuristic

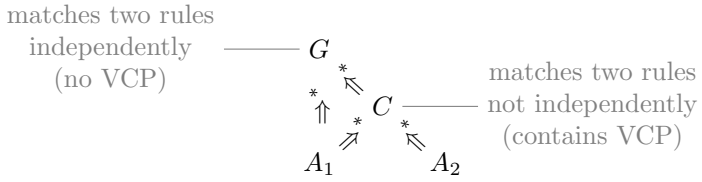


Figure 5.4: Abstraction blocking heuristic issue with not fully concrete VCP

The critical pairs of the grammar that we used for the evaluation where the common subgraph heuristic could be applied all required that the heuristic adds a collapsed rule. This leads to an implementation issue because a collapsed rule needs to be added without a base rule. For simplicity, we decided to implement the heuristic to only add noncollapsed rules and just ignore cases where a collapsed rule needs to be added. Because of this, the heuristic cannot be used for the grammars in the evaluation and therefore it is left out.

Abstraction Blocking Heuristic

This heuristic does not make the grammar confluent but helps to avoid issues that a non-confluent grammar might create. No rules are modified, but when the grammar is used it is required that the abstraction is prevented in certain cases to avoid issues with the non-confluent grammar. This condition has to be chosen in such a way that canonicalizing a graph always creates the same graph, regardless which of the matching rules is used during abstraction.

A simple condition is to just stop the abstraction once the partially canonicalized graph contains a joint graph of a VCP. However, there is an issue with this idea that is shown in Figure 5.4, which shows four graphs and their abstraction relation. The graph C contains the joint graph of the VCP and A_1 and A_2 are fully abstracted graphs. There is also a graph G, which does not contain a VCP but does match two rules for abstraction. The issue arises when graph G is canonicalized. When G is abstracted so that graph C is reached the canonicalization would be halted because it contains the joint graph of a VCP. This would avoid the issue that the result of the canonicalization of C is not deterministic, but it is also possible that G can be abstracted by a different abstraction path. This other abstraction path could, of course, contain another graph with a VCP, which would terminate the canonicalization earlier, but it is not guaranteed that the graph where the canonicalization is halted is isomorphic to C.

A solution to this problem is to only use the abstraction blocking heuristic with VCPs where the joint graph is fully concrete, meaning it contains no nonterminal edges. This ensures that there cannot exist a graph like G. Either G already contains a fully concrete VCP in which case the abstraction is immediately blocked ensuring a deterministic result. If G does not contain a fully concrete VCP, all its abstractions cannot contain a fully concrete VCP because abstracting a graph can only remove nodes and add nonterminal edges.

A drawback of this approach is, that it reduces the abstraction power of the given grammar because abstractions that were possible previously are no longer allowed.

Algorithm 5 Data structure grammar check

Input: *grammar* - An HRG
Output: A boolean indicating whether the input grammar is a data structure grammar.

```
1: function DataStructureGrammarCheck(grammar)
2:     for all rhs ∈ RighHandSides(grammar) do
3:         for all node ∈ Nodes(rhs) do
4:             outgoingSel ← DirectOutgoingSelectors(node)
5:             tentacles ← ConnectedTentacles(node)
6:             tentacleTypes ← GetTentacleTypes(tentacles)
7:             if ¬DisjointSets(tentacleTypes) ∨ outgoingSel ∩ ⋃tentacleTypes ≠ ∅ then
8:                 return false
9:             end if
10:        end for
11:    end for
12:    return true
13: end function
```

5.1.3 Grammar Validity Checks

The grammar validity check allows disregarding grammars that do not fulfill certain conditions. The conditions that are important for Attestor are that the resulting grammar is still a data structure grammar (Section 2.4) and that the grammar is still local concretizable (Section 2.5). As the grammars in Attestor are designed by hand and are just expected to fulfill these conditions there is no check implemented for these properties. The following sections describe algorithms that allow detecting if a grammar satisfies these properties.

Data Structure Grammar Check

A data structure grammar does not allow the creation of two outgoing selectors of the same type at the same node as explained in Section 2.4.

To decide if a grammar fulfills this property it is first necessary to check if there are any conflicts between the outgoing selectors that each rule immediately creates and the selectors that are created by the connected nonterminal edges. The pseudocode in Algorithm 5 based on [4] visualizes how this works in detail. It loops over all nodes in all RHS's of the grammar and then computes the outgoing selectors at this node in the graph. In Line 5, the tentacles of all nonterminal edges connected to the node are computed. Each tentacle can create a different set of outgoing selectors at its connected node. We call the set of possible selectors the tentacle type and this set is computed for all the connected tentacles. If there are two connected tentacles where the tentacle types are *not* disjoint then this means that it is possible that the grammar can create two outgoing selectors of the same type, which means the grammar is not a data structure grammar. It is also necessary to check that there is no collision with the outgoing selectors that are already in the graph. These two checks are done in Line 7 and if a violation has been found, the algorithm returns $false$. If no violation has been found then the given grammar is a data structure grammar and so the algorithm returns *true*.

The tentacle types can be computed recursively. An example grammar rule to demonstrate how this is done is shown in Figure 5.5. When replacing the nonterminal edge A a selector edge is added from node 1 to node 2. Additionally, a nonterminal edge of type A and type B is added, but the edge of type A is reversed. For this rule, it is possible to derive a relation of the tentacle types. For the first tentacle, we get $T(A,1) = \{s\} \cup T(A,2) \cup T(B,1)$ and for the second tentacle, we get $T(A,2) = T(A,1) \cup T(B,2)$. Here $T(A,1)$ refers to the tentacles types of the 1^{st} tentacle of the nonterminal edge A. We see that there is a circular dependency between $T(A,1)$ and $T(A,2)$. Those dependencies can be eliminated by computing the spanning tree of the graph overall dependencies. It is not necessary to compute a fixed point and the circular dependencies can just be removed. So

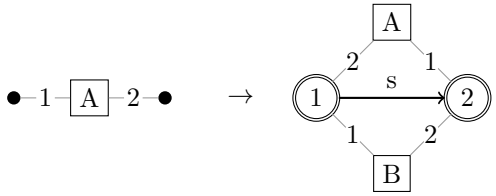


Figure 5.5: Grammar rule for tentacle type example

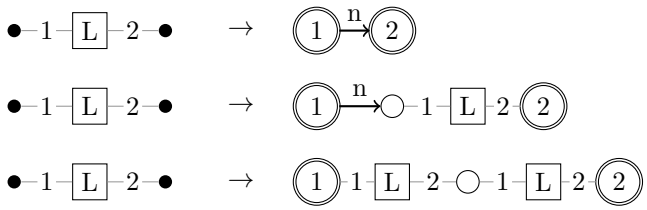


Figure 5.6: Grammar example where the local concretizability check fails

if we assume that the tentacle types of B are $T(B,1) = \{x\}$ and $T(B,2) = \{y\}$ then the tentacle types of A can be computed according to Equations (5.1) and (5.2).

$$\begin{aligned} T(A,1) &= \{s\} \cup T(A,2) \cup T(B,1) \\ &= \{s\} \cup (T(A,1) \cup T(B,2)) \cup T(B,1) \\ &= \{s\} \cup T(B,2) \cup T(B,1) = \{s,x,y\} \end{aligned} \tag{5.1}$$

$$\begin{aligned} T(A,2) &= T(A,1) \cup T(B,2) \\ &= (\{s\} \cup T(A,2) \cup T(B,1)) \cup T(B,2) \\ &= \{s\} \cup T(B,1) \cup T(B,2) = \{s,x,y\} \end{aligned} \tag{5.2}$$

Local Concretizability Check

In Section 2.5, we introduced the property of *local concretizability*. Now we want to present the idea of an algorithm that can detect if the modified grammar is locally concretizable. The algorithm might produce false negatives, meaning a grammar that is local concretizable is not detected as such. However, grammars are never wrongfully detected as locally concretizable.

A modified grammar is guaranteed to be locally concretizable if all external nodes of the added rules are not connected by non-reduction tentacles to any nonterminal. This means that when a nonterminal edge is concretized all selector edges that start at a node that was connected to the nonterminal edge are created immediately and not recursively. In this case, the grammar must be local concretizable because by the definition of non-reduction tentacles there will not be created any outgoing selectors at the external nodes after further rule applications. To compute whether a tentacle is a reduction tentacle or not it is necessary to compute the tentacles types as described in Section 5.1.3.

There might be cases where the check described above fails, but the grammar is still local concretizable. The grammar shown in Figure 5.6 is an example of such a problem. It is local concretizable because the last rule can always be omitted without changing the language of the grammar. The other two rules immediately create the outgoing selector n at external node 1 and there are no selectors created recursively at external node 2. The issue with the local concretizability check is that it is not able to determine that the third rule can be omitted without changing the language, which would be required for local concretizability because it does not create the outgoing selector of type n at the external node 1.

To check if a modified grammar is local concretizable we assume that the unmodified grammar

was local concretizable before the modification. If new rules have been added to the grammar it is only necessary to check if the new rules fulfill the condition that external nodes are not connected to non-reduction tentacles. This allows the user to add rules like in the example of Figure 5.6 for which it is known that the last rule does not violate the condition of local concretizability because the language of this rule is already covered by the first two rules. In case the modified grammar only adds rules to the grammar this does not change the fact that the language of those rules is already covered by other rules so it is not necessary to check those rules. In case a rule of the unmodified grammar is removed in the modified grammar we no longer can make this assumption. Here it is necessary that all rules of the new grammar have to be checked if they fulfill the local concretizability condition. In some cases, it is possible to restrict the set of affected rules of the original grammar so not *all* rules have to be checked, but for simplicity, our algorithm just checks all rules in those cases.

5.2 Implementation

In this section, we discuss how our implementation allows creating different completion algorithms that use different sets of heuristics with different priorities. We give insight into changes to the grammar class of Attestor that are necessary to execute the completion procedure. Lastly, we give details on the implementation of the heuristics that for example influence the order in which the heuristics enumerate the possible grammar modifications.

5.2.1 Completion Algorithm Structure

As described in Section 5.1, the completion algorithm is split into three parts: The completion strategy, the completion heuristics and the validity checks. We implemented those parts to be replaceable to easily test different combinations of those parts and to be able to further extend the completion procedure by adding more heuristics or completion strategies. The completion metric that is used to determine if one grammar should be chosen over another grammar is also exchangeable, even though we just used one metric that is based on the number of VCPs. We call a fixed combination of those parts a *completion algorithm*. It contains exactly one completion strategy, one completion metric, at least one completion heuristic and optionally a set of grammar validity checks. The order in which the completion heuristics are added is important because it determines which heuristic is used first and therefore sets a preference for the heuristics. The order of the grammar validity checks is not important for the result but might influence the runtime. A grammar validity check that takes a long time and rarely fails should not be chosen as the first validity check in case another faster validity check could already eliminate the grammar before.

The listing in Figure 5.7 shows an example of how a custom completion algorithm can be created. Similar to the builder pattern it is possible to set all parts of the completion statement in a single statement. Some classes like the `GreedyCompletion` class allow additional settings. The argument 0 in the `GreedyCompletion` constructor indicates that the number of iterations is not limited. In case a positive value is set, the greedy completion strategy aborts after this number of iterations. When all necessary information is added to the completion algorithm it is possible to execute it with a grammar.

5.2.2 Modified Grammar Format

We implemented a new class to represent grammars to add additional functionality required by our implementation. An issue with Attestor's integrated grammar format is that rules cannot be easily referenced. A rule does not have a simple identifier and if the equality between two rules needs to be checked it requires that both RHS's are compared. Additionally, the relationship between original rules and the collapsed rules is lost. The modified grammar format can reference grammar rules by integer ids and collapsed rules are identified by a combination of the id of the original rule and an id to distinguish its collapsed rules.

```
CompletionState executeCompletionAlgorithm(NamedGrammar grammar) {
    CompletionAlgorithm algorithm = new CompletionAlgorithm("algorithm name")
      .setCompletionStrategy(new GreedyCompletion(0))
      .setCompletionStateLoss(new NumberCriticalPairLoss())
      .addHeuristic(new AddRulesNewNonterminalHeuristic())
      .addHeuristic(new SingleNonterminalRuleAddingHeuristic())
      .addGrammarValidityCheck(new LocalConcretizability())
      .addGrammarValidityCheck(new CheckDataStructureGrammar())
    return algorithm.runCompletionAlgorithm(grammar);
}
```

Figure 5.7: Java code showcasing creation and execution of a completion algorithm

This feature is useful for debugging purposes because it directly shows which rules were used to create a VCP. It is also useful for a user trying to make a grammar confluent to be able to see which of the handwritten rules of the grammar lead to problems.

Another important feature of the modified grammar is the ability to block abstractions, as required by the abstraction blocking heuristic. It changes the way a grammar should be interpreted because the abstraction should not always be executed. A modified version of Attestor's canonicalization procedure was implemented to conform to this abstraction blocking mechanism.

It also allows distinguishing between rules that originate from the initial handwritten grammar and rules that were created by the completion procedure. For the rule restriction heuristic, it allows to mark rules as inactive, which means that the rule is not used for abstraction, but can still be used for concretization.

5.2.3 Efficient Enumeration by Usage of Iterators

A completion heuristic can create many different modified grammars from which the completion strategy chooses one as the next grammar. It is inefficient in regards to the runtime and the memory usage to compute all those possible grammars every time the heuristic is invoked. By using iterators, the algorithm only computes the grammars as they are needed by the completion strategy and if a loop is aborted the remaining grammars are never computed. If the completion strategy has decided against a grammar, there is no reference to the grammar object and it can be removed by the garbage collector.

The implementation of iterators in Java is quite verbose requiring to create a new class for each iterator that implements a `hasNext()` and a `next()` method. Therefore we implemented a simplified abstract class called `SimpleIterator` which implements the interface methods of the iterator class and just requires a single method `computeNext()` to be implemented by its subclasses. This makes it easier to write anonymous classes in the different heuristic implementations to implement subclasses of the `SimpleIterator`.

5.2.4 Heuristic Setup

It is possible to specify settings for different heuristics when creating a completion algorithm. All heuristics have a default setting which is used in the evaluation of the completion procedure. In this section, we present the settings we implemented and provide implementation details that determine *which* grammars are enumerated by the heuristics and in what order they are enumerated.

The rule restriction heuristics has two boolean parameters that allow enabling the reactivation of rules and a boolean to prevent the deactivation of original grammar rules. The reactivation of rules is disabled by default to ensure termination and the deactivation of original rules is prevented as this would modify the language too much.

The new nonterminal heuristic has two integer parameters that specify the minimum number of externals for the new nonterminal and the maximum number of nonterminals. The default

value of both integers is 1, which means that only a nonterminal of rank 1 is used. The heuristic tried all possible combinations that the new nonterminal can connect to the two RHS's. If the bounds on the number of external nodes include multiple integers, the heuristic first enumerates modified grammars with nonterminals of the lower rank and when all combinations have been tried, it enumerates the combinations for a nonterminal of the higher ranks. For two RHS's with sets of nodes N_1 and N_2, the combinations for rank n are computing by enumerating all possible combination of subsets S_1 and S_2 where $S_1 \subseteq N_1$, $S_2 \subseteq N_2$ and $|S_1| = |S_2| = n$. For one such combination of subsets, the nonterminal connects to the nodes in S_1 one of the created rules and connects to the nodes in S_2 in the other created rule. We do not enumerate different combinations that the nonterminal can connect to the nodes of S_1 and S_2.

The *join generated nonterminal* heuristic does not have any settings. It iterates over all VCPs and if the fully abstracted graphs of the VCP are just handles of generated nonterminals of the same rank, it returns a modified grammar where those nonterminals are joined.

The *single nonterminal* heuristic also does not require any settings. In case one fully abstracted graph of a VCP is a handle, it is chosen as the LHS of the new rule and therefore the rank is fixed. The heuristic tries all possible combinations of external nodes while not including different orders of external nodes and it does not create collapsed rules. Because we do not compute collapsed rules the number of nodes of the other graph has to be at least as large as the rank of the nonterminal.

The *handle with common subgraph* heuristic has no settings. It iterates over all VCPs and then tries to choose every nonterminal of both fully abstracted graphs as LHS for a new rule. A subgraph check from Attestor is used to determine the common subgraph in the other fully abstracted graph. It would be possible that there are multiple matches for the common subgraph, but the implementation of the subgraph check does not allow to iterate over all possible matches. Therefore the heuristic is not able to returns all possible combinations. The heuristic also does not create collapsed rules and just omits modifications that would require a collapsed rule.

The *abstraction blocking* heuristic always just returns one modified grammar where the abstraction is blocked for all fully concrete graphs that are a joint graph of a VCP. Heuristics are intended to modify as little as possible of a grammar in each step, so according to this requirement, it would make sense to just block a single graph in one step. However, the way the completion strategy is implemented, the same heuristic is executed again if there was any progress and therefore would lead to the same result. One drawback of this implementation is that a supergraph of a graph that is already in the set of blocked graphs is added. This supergraph increases the number of subgraph checks required during abstraction, but it would already be blocked because the subgraph is already included.

5.3 Evaluation

In this section, we evaluate the performance of our grammar completion implementation. We are not aware of any other tool that provides automatic completion for any kind of graph grammar so a comparison with other work is not included. We first provide information about the different test setups and how the evaluation was executed. Then results of the runtime, the remaining number of VCPs and the modifications to the grammars for the test cases are presented and discussed.

5.3.1 Methodology

For the evaluation, we created multiple different completion algorithms that consist of different combinations of heuristics used and grammar validity checks. All these algorithms are shown in Table 5.1, where we assigned each algorithm a number (A1-A11). The table specifies the priority of the heuristics and validity checks, where 1 means that this part is executed first. The numbering starts at 1 again for the grammar validity checks because they are part of a different phase of the complete procedure.

The algorithms A1 to A5 are intended to test the performance of a single heuristic. In the case of A4, the *new nonterminal* heuristic is used additionally because just using the *join new*

Heuristic / Validity Check	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11
rule restriction	1							5	5	2	2
abstraction blocking		1						1	1	1	1
new nonterminal			1	1	1	1	1	2	2	3	3
join new nonterminals				2		2	2	3	3	4	4
direct handle					2	3	3	4	4	5	5
local concretizability check						1		1		1	
datastructure grammar check	1	1	1	1	1	2	1	2	1	2	1

Table 5.1: Overview of different combinations of heuristics used for completion

nonterminal heuristic without actually creating new nonterminals does not make sense. The *direct handle* heuristic alone is also not useful because the case that a VCP contains a fully abstracted graph that is just a handle does not occur that often. The main difference between the algorithms A8 to A11 is that the priority of the heuristics is changed and that two of those algorithms do not include the local concretizability check.

The time measurement is done similar to the confluence detection evaluation, where the `ThreadMXBean.getCurrentThreadCpuTime()` method of Java is used in a stopwatch object. As the completion procedure takes more time to complete as the confluence detection and because we have a larger number of test cases resulting from the different combinations of completion algorithms to test it was necessary to run multiple tests in parallel to speed up the evaluation. For the parallel execution, it was necessary to modify the code of Attestor because the usage of a static variable in Attestor's implementation of the VF2 algorithm, used for the isomorphism check, creates race conditions when running in parallel. This issue was fixed by using Javas `ThreadLocal` class for those static variables. The benchmark code uses a thread pool to which all different benchmark setup combinations where enqueued once. To avoid issues with Java caching results the benchmark was then executed multiple times to be able to average the test results.

5.3.2 Results

In this section we first present runtime results for all combinations of completion algorithms A1 to A11 with the grammars SimpleDLL, DLList, InTree, LinkedTree1, and LinkedTree2. These grammars were also used for the confluence detection and the data structures they represent is explained in Section 4.3. The other grammars explained in this section were not used because they already are confluent.

After discussing the runtime results we then provide insight in the way the grammars are modified look and discuss their usefulness for analysis by Attestor.

Additional evaluation results can be found online[1]. We provide PDFs that show the rules of each grammar before and after the completion procedure for all combinations of grammars and completion algorithms that we evaluated. We also provide PDFs that show the critical pairs of the grammar before and after the completion procedure.

Runtime Analysis

The runtime results of all algorithms that we evaluated are shown in Table 5.2. For easier readability a copy of Table 5.1 is included to see what heuristics are used by the completion algorithms A1-A11. The table shows the number VCPs the number of rules (R) including collapsed rules and the time (T) of the specific benchmark run. The *Init* column shows the initial number of VCPs and rules before the completion algorithm has been executed. A grammar is confluent if the number of VCPs is zero and for better visibility, those zeroes are marked with a red background. The runtime

[1] https://github.com/Johannes-S/confluence — Hash: 3149498bea27cf3cbb5b4c6682383b6b2d519faa

Grammar		Init	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11
SimpleDLL	VCP	1	1	1	1	1	0	1	0	1	0	1	0
	R	2	2	2	2	2	3	2	3	2	3	2	3
	T	-	1 ms	1 ms	18 ms	18 ms	28 ms	28 ms	27 ms	28 ms	26 ms	26 ms	26 ms
DLList	VCP	75	3	65	54	40	54	62	40	59	2	59	2
	R	40	21	40	70	70	70	58	70	48	56	48	25
	T	-	2 min	739 ms	50 min	49 min	50 min	36 min	50 min	27 min	33 min	26 min	3 min
InTree	VCP	23	11	23	22	22	21	23	21	11	2	11	0
	R	12	6	12	14	14	15	12	15	6	9	6	8
	T	-	2 s	202 ms	16 s	16 s	16 s	9 s	16 s	12 s	15 s	9 s	5 s
LinkedTree1	VCP	5	1	5	1	1	1	1	1	0	0	0	0
	R	10	6	10	18	18	18	18	18	17	17	8	8
	T	-	384 ms	71 ms	779 ms	778 ms	733 ms	2 s	623 ms	2 s	536 ms	419 ms	279 ms
LinkedTree2	VCP	221	27	120	132	132	44	98	44	65	0	65	0
	R	20	5	20	188	188	265	178	265	120	171	120	19
	T	-	31 s	500 ms	48 h	48 h	50 h	28 h	50 h	25 h	48 h	25 h	2 min

Heuristic / Validity Check	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11
rule restriction	1							5	5	2	2
abstraction blocking		1						1	1	1	1
new nonterminal			1	1	1	1	1	2	2	3	3
join new nonterminals				2		2	2	3	3	4	4
direct handle					2	3	3	4	4	5	5
local concretizability check						1		1		1	
datastructure grammar check	1	1	1	1	1	2	1	2	1	2	1

Table 5.2: Runtime overview of different combinations of grammars and completion heuristics

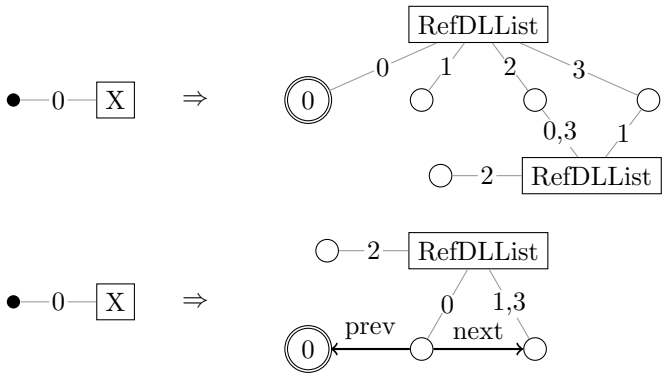


Figure 5.8: Example of generated rules in completion algorithm A8 for the DLList grammar

varies greatly between the different benchmark settings so the time values are also color-coded to visualize the different magnitudes.

The grammars LinkedTree2, SimpleDLL, and DLList do not fulfill the local concretizability condition that we check. The existing rules are only checked for this condition if a rule is removed as explained in Section 5.1.3. Therefore grammars that do not fulfill the local concretizability condition cannot make use of the rule restriction heuristic because if a rule is removed, all rules need to be checked and the local concretizability check fails. In algorithm A10 the local concretizability is enabled and the rule restriction heuristic has a high priority so this effect can be overserved when comparing the number of critical pairs that can be removed by the completion procedure. For the DLList and LinkedTree2 grammar, the number of remaining VCPs is quite high with 65 VCPs for the LinkedTree2 grammar and 59 VCPs for the DLList grammar.

The runtime generally correlates with the number of grammar rules and the number VCPs. This seems reasonable because the more grammar rules there are the more likely it is that two grammar rules lead to a VCP. A completion procedure is expected to run longer if there are more VCPs because there are more issues to fix.

The results for the LinkedTree1 grammar are interesting because algorithm A1 and A3 are both able to reduce the number of critical pairs to 1, but only the combined algorithms A8-A11 can make the grammar confluent. Algorithm A1 uses just rule restriction and A3 uses just the *new nonterminal* heuristic. This shows the significance of using multiple heuristics to achieve confluence.

In many cases, the usage of the rule restriction heuristic with a higher priority leads to finding a confluent grammar faster. The LinkedTree2 grammar is made confluent after just 2 minutes with algorithm A11 in comparison to 48 hours using algorithm A9. In the case of the InTree grammar, the lower priority of the rule restriction heuristic even prevents the completion procedure from making the grammar confluent as there are still 2 VCPs remaining when algorithm A9 is used.

Unfortunately, no completion algorithm was able to achieve confluence for the DLList grammar. Here the rule restriction heuristic is very effective by reducing the number of VCPs from 75 to 3, but the combined algorithms are only able to further reduce this number to 2.

Completed Grammars

The completion procedure creates grammars that are not guaranteed to be language equivalent to the original grammar and we do not have a simple way to compare the actual quality of the modified grammars. This makes it difficult to compare different completion algorithms because a completion algorithm that takes longer, but creates a confluent grammar that is closer to the original grammar would be preferable even if its runtime is longer. Attached to this thesis are the PDFs that show the critical pairs and grammars before and after running the completion algorithms. The readme file in the root directory contains information about the folder structure to find the correct PDF. In this section, we want to present a few examples of how the grammars are modified to better understand the changes made by the strategies.

In Figure 5.8, two rules are shown that are generated by completion algorithm A8 for the DLList grammar. A new nonterminal X has been created, which maps to the two specified graphs. Just for concretization, the introduced rules do not weaken the expressiveness of the grammar because the nonterminal X would not occur in a graph that was used before. However, when abstracting a graph that matches one of the RHS's then the connection which of the two RHS's was contained originally is lost. The structure of the RHS's is very complex and such a graph would rarely occur in practice by transforming an actual doubly-linked list. Such complex rules that are very unlikely to occur during analysis are desirable because they can exist to make the grammar confluent without restricting the capability of the grammar to be used for analysis in Attestor.

6 Conclusion and Future Work

In this chapter we first give a summary of the presented algorithms. We then discuss some further changes that could further improve the usability of the completion procedure.

6.1 Summary

In this thesis, we present a confluence detection algorithm that can compute the critical pairs of a hyperedge replacement grammar (HRG). To compute all critical pairs we efficiently enumerate the overlappings and prune irrelevant overlappings early on, which results in a fast enumeration of the critical pairs. The algorithm computes the joinability of the critical pairs to determine if the grammar is confluent and can decide confluence for the grammars that we consider. The runtime of our implementation is magnitudes faster than that of AGG. A reason for the speed improvement is that AGG is implemented for a more general graph transformation system.

Based on the confluence detection algorithm we implemented a completion algorithm. The algorithm takes an HRG, which is not confluent and tries to make it confluent. The format of HRGs limits what kind of rules are possible. The completion algorithm uses a collection of different heuristics that have different advantages and drawbacks. It ensures that the resulting grammar is still a data structure grammar, which is required by Attestor. Additionally, it is possible to only accept grammars that are guaranteed to be locally concretizable. The easily extendable structure of the completion algorithm makes it simple to add new heuristics, completion loss functions, completion strategies, and grammar validity checks. Depending on the grammar the runtime of the completion procedure varies from a few milliseconds to multiple days. The result of the completion procedure is in the best case a confluent grammar, or if the algorithm was not able to achieve confluence a grammar with a fewer number violating critical pairs (VCPs). The grammar that the completion algorithm produced can then be checked by the developer by using the included TikZ output. The grammar can then be further modified by hand or immediately used in an analysis of Attestor.

6.2 Future Work

In this section, we discuss features that are not implemented in the presented algorithms, but which have the potential to return even better results. They could improve the quality of the completed grammars to be more useful in the task of program analysis.

6.2.1 Language Equivalence

The implementation of the completion procedure does not consider language equivalence, which can be a problem, because it can reduce the usefulness of the grammar. Even if exact language equivalence is not necessary it is still beneficial if the language of the modified grammar does not differ too much. There is always a trivial confluent grammar where all critical pairs just map to the same nonterminal. Our implementation addresses this issue by only modifying the grammar in small steps and only keeping modifications that are necessary to reach a confluent grammar. This approach, however, is no guarantee that a useless trivial confluent grammar is returned, even though a more useful confluent grammar exists. Therefore, it is desirable to be able to look at the language of the modified grammar.

Language equivalence is an undecidable problem. However, it is possible to implement incomplete methods that allow in some cases to determine that the language of two grammars are equivalent

or not. Showing that the grammars are not equivalent can be done by finding a graph that can be derived by one grammar but not the other. It would be possible to use such an incomplete method to either only allow grammars for which the equivalence is guaranteed, which might eliminate possible grammars which would be equivalent but the equivalence cannot be proven. Alternatively, the completion algorithm could only reject grammars for which a counterexample to equivalence has been found, which would increase the confidence that the grammars are equivalent but does not guarantee that the grammars are equivalent.

Our implementation of the completion algorithm allows an easy exchange of the loss function by which the grammar is chosen. The quality of the resulting grammar could be improved by including a metric that prefers grammars whose language is closer to the original grammar. The development of such a metric could, however, be very difficult.

6.2.2 Improved Completion Strategies

The completion strategy that we implemented is a simple greedy algorithm. The advantage is a fast and simple implementation, but the greedy algorithm might not find the best solution because a choice that was made once cannot be undone. It is possible to use heuristics that can reverse changes by deleting rules that were added or adding rules that have been removed from the original grammar. This would also require to modify the completion loss function to prefer grammars with fewer changes in case they do not reintroduce the critical pairs.

A possibly better solution would be a different completion strategy implementation based on a search algorithm like A*. This would allow the completion algorithm to apply multiple heuristics after the other that don't improve the loss value of the grammar after the first heuristic applications, but at the end lead to a better grammar than only applying heuristics that result in a better grammar after *each* step.

6.2.3 Iterative Completion Refinement

The completion algorithm was implemented for Attestor which is a static program analysis tool. We cannot ensure that the language of the completed grammar returned by the completion algorithm is equivalent to the original grammar. This means that the analysis might fail with the modified grammar but a different completed grammar could result in a succeeding analysis.

The completion procedure could be improved by using feedback from the analysis procedure. This could be implemented as simple as a loop that first runs the completion algorithm then uses the resulting grammar in the analysis. In case of failure, the algorithm could go back to the completion algorithm to compute a different completed grammar to use in the analysis. A more advanced approach can be implemented that narrows down the modifications of the grammar that lead to the failure to return more targeted completed grammars that have a higher probability of success.

Bibliography

[1] Hannah Arndt, Christina Jansen, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. Let This Graph be your Witness! An Attestor for Verifying Java Pointer Programs. In *International Conference on Computer Aided Verification*, pages 3–11. Springer, 2018.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[3] Jonathan Heinen. *Verifying Java Programs - A Graph Grammar Approach*. PhD thesis, RWTH Aachen University, 2015.

[4] Jonathan Heinen, Christina Jansen, Joost-Pieter Katoen, and Thomas Noll. Juggrnaut: Using Graph Grammars for Abstracting Unbounded Heap Structures. *Formal Methods in System Design*, 47(2):159–203, 2015.

[5] Ivaylo Hristakiev. *Confluence Analysis for a Graph Programming Language*. PhD thesis, University of York, UK, 2018.

[6] Gérard Huet. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems. In *18th Annual Symposium on Foundations of Computer Science*, pages 30–45. IEEE, 1977.

[7] Christina Jansen. *Static Analysis of Pointer Programs - Linking Graph Grammars and Separation Logic*. PhD thesis, RWTH Aachen University, Germany, 2017.

[8] Donald E. Knuth and Peter B. Bendix. Simple Word Problems in Universal Algebras. In *Automation of Reasoning*, pages 342–376. Springer, 1983.

[9] Christoph Matheja, Christina Jansen, and Thomas Noll. Tree-Like Grammars and Separation Logic. In *APLAS*, volume 9458 of *Lecture Notes in Computer Science*, pages 90–108. Springer, 2015.

[10] Detlef Plump. Hypergraph Rewriting: Critical pairs and Undecidability of Confluence. *Term Graph Rewriting: Theory and Practice*, 15:201–213, 1993.

[11] Detlef Plump. Confluence of Graph Transformation Revisited. In *Processes, Terms and Cycles: Steps on the Road to Infinity*, pages 280–308. Springer, 2005.

[12] Detlef Plump. Checking Graph-Transformation Systems for Confluence. *Electronic Communications of the EASST*, 26, 2010.

[13] Grzegorz Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. world Scientific, 1997.

[14] Gabriele Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *AGTIVE*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer, 2003.

[15] Till Tantau. Graph drawing in TikZ. In *International Symposium on Graph Drawing*, pages 517–528. Springer, 2012.