A Static Analysis of Non-Confluent Triple Graph Grammars for Efficient Model Transformation

Anthony Anjorin[1]*, Erhan Leblebici[1], Andy Schürr[1], and Gabriele Taentzer[2]

[1]Technische Universität Darmstadt,
Real-Time Systems Lab, Germany
{anjorin,leblebici,schuerr}@es.tu-darmstadt.de

[2]Philipps-Universität Marburg,
Fachbereich Mathematik und Informatik, Germany
taentzer@mathematik.uni-marburg.de

Abstract. Triple Graph Grammars (TGGs) are a well-known bidirectional model transformation language. All actively developed TGG tools pose restrictions to guarantee efficiency (polynomial runtime), without compromising formal properties. Most tools demand *confluence* of the TGG, meaning that a choice between applicable rules can be freely made without affecting the final result of a transformation. This is, however, a strong restriction for transformations with inherent degrees of freedom that should not be limited at design time. eMoflon is a TGG tool that supports *non-confluent* TGGs, allowing different results depending on runtime choices. To guarantee efficiency, nonetheless, a local choice of the next source element to be translated, based on source context dependencies of the rules, *must not* lead to a *dead end*, i.e., to a state where no rule is applicable for the currently chosen source element to be translated, and the transformation is not yet complete. Our contribution in this paper is to formalize a corresponding property, referred to as *local completeness*, using graph constraints. Based on the well-known transformation of constraints to application conditions, we present a static analysis that guarantees dead end-freeness for non-confluent TGGs.

Keywords: bidirectionality, triple graph grammars, static analysis

1 Introduction and Motivation

Model synchronization is a crucial task in numerous application domains. In a current research project, we have investigated and implemented a tool for synchronizing two textual languages used in the domain of *Concurrent Manufacturing Engineering* (CME). The tool[1] is able to propagate changes incrementally from documents in one language to documents in the other, thus enabling a concurrent engineering workflow.

* The project on which this paper is based was funded by the German Federal Ministry of Education and Research, funding code 01IS12054. The authors are responsible for all contents.

[1] A screencast demonstrating the tool is available at www.emoflon.org

Triple Graph Grammars (TGGs)[13] are a formally founded rule-based bidirectional model transformation language, and were used in the CME research project to realize the synchronization of models formulated in different modelling languages. We have identified TGGs to be the typical performance bottle-neck in such transformation chains [11] meaning that improving the *efficiency* (i.e., achieving polynomial runtime in model size) of TGG-based transformations is a current and crucial challenge. To the best of our knowledge, all TGG tools strive to guarantee efficiency by posing certain restrictions on the class of supported TGGs. The specification for the synchronization tool in the CME research project consists of about a 100 TGG rules, which means that manually checking all such restrictions is practically infeasible. For specifications of this size and larger, an automated and comprehensive static analysis of all required restrictions becomes crucial.

A common strategy to achieve efficiency is to demand *confluence*, meaning that choices between applicable TGG rules do not influence the final result of the transformation. This improves efficiency as wrong choices that might lead to *dead ends*, i.e., states where no rule is applicable but the transformation is not yet complete, are no longer possible [8].

In many application scenarios such as for the CME research project, however, the required transformations often have an inherent degree of freedom, which cannot always be restricted at design time to ensure confluence. In the CME research project, for example, the end-user (or a configuration module) must guide the synchronization appropriately, making choices based on case-by-case preferences. Adjusting the underlying TGG and rebuilding the synchronization tool for each possible set of choices is simply infeasible.

To support non-confluent TGGs and nonetheless ensure efficiency without compromising formal properties, the TGG tool eMoflon (`www.emoflon.org`) [12] determines a sequence in which source elements can be translated based only on source context dependencies of the TGG rules. This strategy has been shown to be efficient in [9], if a local choice of the next source element to be translated cannot lead to a dead end in the transformation. "Local" means that the entire source model is never searched globally for the next translatable element.

There is currently no static analysis for this required property of non-confluent TGGs referred to as *local completeness* [9], meaning that an exception is thrown whenever the condition is violated at runtime. As only suitable tests can reveal this, local completeness violations are currently one of the most common and frustrating mistakes made by eMoflon users, especially beginners.

Our main contribution in this paper is to formulate the condition for local completeness as a set of graph constraints, thus providing a constructive formalization and a static analysis for local-completeness of non-confluent TGGs. We apply well-known techniques, e.g., for transforming constraints to sufficient and necessary application conditions [4], which have already been shown in previous work [1,8] to be applicable in general to TGGs.

The paper is structured as follows: in Sect. 2 we provide a running example and recall basic definitions and results for TGGs. Our main contribution is presented in Sect. 3, providing a static analysis for non-confluent TGGs. Section 4 gives an overview of related approaches, while Sect. 5 concludes the paper with a brief summary and discussion of future work.

2 Running Example and Preliminaries

As a running example, we consider a *Platform-Independent Model* (PIM) to *Platform-Specific Model* (PSM) transformation from the CME domain. An example is depicted in Fig. 1. The PIM is represented as a *Cutter Location Source* (CLS) file, which specifies the manufacturing process as a series of operations. CLS files can be executed in a simulator that visualizes the specified manufacturing process. The PIM is used to generate machine-specific *G-code*, depicted as the PSM to the right of Fig. 1. G-code programs can be executed on appropriate machines that realize the manufacturing process. In practice, G-code programs are sometimes optimized manually. For example, the sequence of operations used to move the machine to its initial position can be shortened for a particular machine and set-up. Manual updates to G-code programs that can be expressed on the PIM level must be propagated back to CLS as they would otherwise be overwritten and lost during code generation. This propagation must be *incremental* as CLS files contain information, which is discarded during code generation and cannot be regained from G-code.

In our example, we consider only the most basic operation used to move the tip of the current machine. It is specified in two different ways: (i) the machine performs a linear interpolation between its current and the target location maintaining a constant *feedrate* (speed), and (ii) the machine is free to realize the movement to be as *rapid* as possible. In the CLS syntax, this basic operation is specified via a GOTO/ X, Y, Z operation, where (X, Y, Z) are the coordinates of the target location. As a safety feature, an additional RAPID command is required to indicate that the next GOTO is to be executed in rapid mode. After such a "rapid" GOTO, the machine reverts to the default feedrate mode.

In G-code, a series of "G" switches are used to influence how the current *and all following operations* are executed by the machine. The switches G0 and G1 correspond to rapid and feedrate mode, respectively. In contrast to the CLS

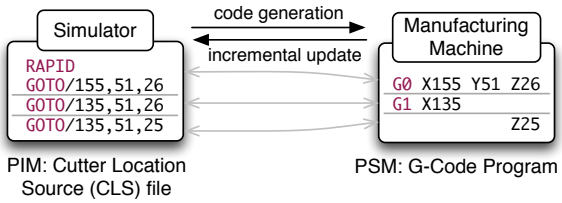


Fig. 1. An example of a PIM-to-PSM transformation in the CME domain

format, G-code is optimized for efficient interpretation, only requiring the actual changes from the previous line to be specified on each new line. On the second line, for example, G1 X135 is equivalent to G1 X135 Y51, G1 X135 Z26, or G1 X135 Y51 Z26. The G-code program depicted in Fig. 1 is, therefore, only one of 32 correct G-code programs! Is this flexibility required? Why not enforce the most efficient G-code program as depicted in Fig. 1? The reason is that efficiency is sometimes traded for maintainability of the templates used to generate G-code programs, i.e., values are repeated so that transformation templates can be reused in a different context. Before we specify TGG rules that appropriately capture this degree of freedom of the transformation, we have to establish a basic understanding of models, metamodels, and rule-based model transformation.

2.1 Consistency Specification with Triple Graph Grammars

In line with the algebraic formalization according to [3], *models* and *metamodels* are formalized as typed graphs and type graphs, respectively:

Definition 1 (Graph and Graph Morphism).
A graph $G = (V, E, s, t)$ *consists of finite sets* V *of nodes and* E *of edges, and two functions* $s, t : E \to V$ *that assign each edge source and target nodes.*
A graph morphism $h : G \to G'$*, with* $G' = (V', E', s', t')$*, is a pair of functions* $h := (h_V, h_E)$ *where* $h_V : V \to V'$, $h_E : E \to E'$, $h_V \circ s = s' \circ h_E \wedge h_V \circ t = t' \circ h_E$.

Definition 2 (Typed Graph and Typed Graph Morphism).
A type graph *is a graph* $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$.
A typed graph *is a pair* $(G, type)$ *of a graph* G *together with a graph morphism* type*:* $G \to TG$*. Given* $(G, type)$ *and* $(G', type')$*,* $g : G \to G'$ *is a* typed graph morphism *iff* $type = type' \circ g$.
$\mathcal{L}(TG) := \{G \mid \exists\, type : G \to TG\}$ *denotes the set of all graphs of type* TG.

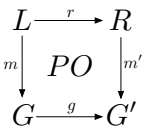
Remark 1 (The Category of Typed Graphs)
Typed graphs and typed graph morphisms form a category **Graphs** *with the set* $\mathcal{M}$ *of injective typed graph morphisms (cf. [3]).*

Remark 2 (Attributed Typed Graphs with Inheritance)
As we formulate our definitions and theorems on the level of typed graphs and typed graph morphisms, they can be extended to attributed *typed graphs with* node type inheritance *in a straightforward manner.*

Next, *rule*-based model transformation is formalized, where rules have pre- and post-conditions with *constraints* that either hold globally for all models, or are used to guard rule application (*as application conditions*). As we only require creating (monotonic) rules for TGGs, the following definitions from [3] are thus simplified appropriately.

Definition 3 (Rule, Graph Grammar, and Derivation).
A rule *is a typed graph morphism* $r : L \to R \in \mathcal{M}$*, where* TG *is a type graph and* $L, R \in \mathcal{L}(TG)$*. A* graph grammar *is a pair* $GG = (TG, \mathcal{R})$ *of a type graph* TG *and a finite set* $\mathcal{R}$ *of rules.*

A direct derivation $G \overset{r@m}{\Longrightarrow} G'$ *(or* $G \overset{r}{\Longrightarrow} G'$*) is given by a pushout in* ***Graphs*** *(cf. diagram to the right).*



A derivation $G \overset{*}{\Longrightarrow} G'$ *of length* $n \geq 0$ *in* $GG = (TG, \mathcal{R})$ *is a sequence of* n *direct derivations* $G \overset{r_1}{\Longrightarrow} G_1 \overset{r_2}{\Longrightarrow} \cdots \overset{r_n}{\Longrightarrow} G'$*, with* $r_1, r_2, \cdots, r_n \in \mathcal{R}$*. In case of length* $n = 0$*, we have* $G' = G$*.*
$\mathcal{L}(GG, G_\emptyset) := \{G \in \mathcal{L}(TG) \mid G_\emptyset \overset{*}{\Longrightarrow} G\}$ *denotes the language generated by a graph grammar* GG*, where* $G_\emptyset \in \mathcal{L}(TG)$ *denotes the start typed graph.*
$\mathcal{L}(GG) := \mathcal{L}(GG, \emptyset)$*, where* $\emptyset$ *is the empty typed graph.*

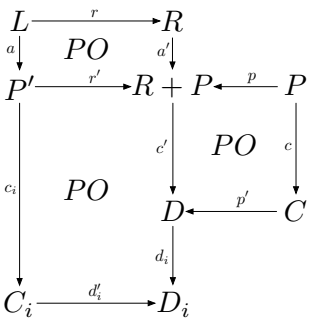
Definition 4 (Conditional Constraints).
A conditional constraint *c is a typed graph morphism $c : P \to C$.*
For conditional constraints c_i with $i \in I$ for some index set I, $\vee_{i \in I} c_i$ is also a conditional constraint.
A typed graph G satisfies a conditional constraint c, denoted by $G \models c$, if either $c : P \to C$ and $\forall p : P \to G \in \mathcal{M}$, $\exists q : C \to G \in \mathcal{M}$ such that $q \circ c = p$, or $c = \vee_{i \in I} c_i$ and $\exists i \in I : G \models c_i$.
A graph grammar $GG = (TG, \mathcal{R})$ satisfies a conditional constraint c, denoted by $GG \models c$, if for all derivations $G \overset{*}{\Longrightarrow} G'$ with $G \in L(GG, G_\emptyset)$, $G' \models c$.

Definition 5 (Conditional Application Conditions).
A conditional application condition *over a typed graph L is a pair $ac = (a, \vee_{i \in I} c_i)$, where $a : L \to P$ and $c_i : P \to C_i$ with $i \in I$, for some index set I.*
A typed graph morphism $m : L \to G \in \mathcal{M}$ satisfies a conditional application condition ac, denoted by $m \models ac$, if $\forall p : P \to G \in \mathcal{M}$ with $p \circ a = m$, $\exists i \in I$ and $q_i : C_i \to G \in \mathcal{M}$ such that $q_i \circ c_i = p$.
A conditional application condition for $r : L \to R$ is a conditional application condition over L.
A rule r with a set of conditional application conditions $\mathcal{AC}$ is denoted by $(r, \mathcal{AC})$.
A graph grammar $GG = (TG, \mathcal{R})$ is a graph grammar without conditional application conditions, if all rules in $\mathcal{R}$ do not have conditional application conditions.
A conditional application condition ac is trivial*, i.e., always satisfied, if $\exists i \in I$ such that $c_i : P \to P$ is the identity.*
A Negative Application Condition (NAC) *$ac = (a, \vee_{i \in I} c_i)$ is a conditional application condition where $a : L \to P$ and I is the empty index set.*
A NAC is, therefore, simply denoted by a typed graph morphism $a : L \to P$.

Given a TGG and a set of global constraints that must hold for all models, appropriate application conditions can be automatically generated for every TGG rule, ensuring that the constraints are never violated.

Fact 1 (Construction of Application Conditions from Constraints)
Given a graph grammar $GG = (TG, \mathcal{R})$ without conditional application conditions, and a set $\mathcal{C}$ of conditional constraints.
There is a construction A producing a set of rules with conditional application conditions $R' = A(GG, \mathcal{C}) = \{(r, \mathcal{AC}) \mid r \in \mathcal{R}\}$ s.t. $\forall c \in \mathcal{C} : GG' \models c$, with $GG' = (TG, \mathcal{R}')$. The constructed conditional application conditions are sufficient and necessary.

Proof. This is a special case of Thm. 7.23 in [3], proven on the basis of adhesive HLR categories, here for conditional constraints and monotonic rules.
The diagram to the right shows the main steps of the construction, namely: (1) constructing all possible gluings $R+P$ of the right-hand side R of the rule and the premise P of each constraint, (2) producing for each gluing a post-condition via a pushout, where D_i represents all possible further gluings of elements in D, and finally (3) constructing a pre-condition from the post-condition by reversing the application of the rule (determining pushout complements P' and C_i). If this is not possible (i.e., a pushout complement does not exist) then the post-condition does not result in an equivalent pre-condition.



The central idea with TGGs [13,9] is to specify the consistency of source and target models by providing rules that define a language of consistent source and target models, connected by a *correspondence* model. Rules in a TGG thus describe the simultaneous evolution of *triples* of source, correspondence and target models, from which various operational transformations such as forward/backward transformations can be automatically derived. In the following, we denote *typed triple graphs* with single letters, e.g., G, which consist of typed graphs with an index $X \in \{S, C, T\}$, e.g., G_S, G_C, G_T.

Definition 6 (Typed Triple Graph and Typed Triple Morphism).
A triple graph $G = G_S \overset{\sigma_G}{\leftarrow} G_C \overset{\tau_G}{\rightarrow} G_T$ *consists of graphs* G_S, G_C *and* G_T*, and graph morphisms* $\sigma_G : G_C \to G_S$*,* $\tau_G : G_C \to G_T$*.*
A triple morphism $g = (g_S, g_C, g_T) : G \to G'$*,* $G' = G'_S \overset{\sigma_{G'}}{\leftarrow} G'_C \overset{\tau_{G'}}{\rightarrow} G'_T$
is a triple of graph morphisms $g_X : G_X \to G'_X, X \in \{S, C, T\}$*,*
s.t. $g_S \circ \sigma_G = \sigma_{G'} \circ g_C$ *and* $g_T \circ \tau_G = \tau_{G'} \circ g_C$*.*
Given a triple graph $TG = TG_S \overset{\sigma_{TG}}{\leftarrow} TG_C \overset{\tau_{TG}}{\rightarrow} TG_T$ *called* type triple graph*, a* typed triple graph *is a pair* $(G, type)$ *of a triple graph* G *and triple morphism* $type : G \to TG$*.*
Analogously to Def. 2, $\mathcal{L}(TG)$ *denotes the set of all triple graphs of type* TG*.*
Given $(G, type), (G', type') \in \mathcal{L}(TG)$*, a* typed triple morphism $g : G \to G'$ *is a triple morphism such that* $type = type' \circ g$*.*
The source-correspondence graph $sc(G)$ *of typed triple graph* $G_S \overset{\sigma_G}{\leftarrow} G_C \overset{\tau_G}{\rightarrow} G_T$ *is defined as* $G_S \overset{\sigma_G}{\leftarrow} G_C \overset{\tau_G}{\rightarrow} Im(\tau_G)$*, where* $Im(\tau_G) \subseteq G_T$ *is the codomain of* τ_G*.*
Given $g : G \to G'$ *the* source-correspondence morphism $sc(g) : sc(G) \to sc(G')$ *is defined by* $sc(g) = (g_S, g_C, Im(g_C))$ *with* $Im(g_C) = \tau_{G'} \circ g_C \circ {\tau_G}^{-1}$
(${\tau_G}^{-1}$ *exists as it is bijective).*

Example 1. Figure 2 depicts a possible type triple graph and a typed triple graph, representing the consistent pair of PIM and PSM models of our running example (Fig. 1). A UML-like syntax is used and attributes (for coordinates in the running example) are excluded for presentation purposes. Finally, the types of edges in the typed triple graph are omitted as they can be uniquely determined

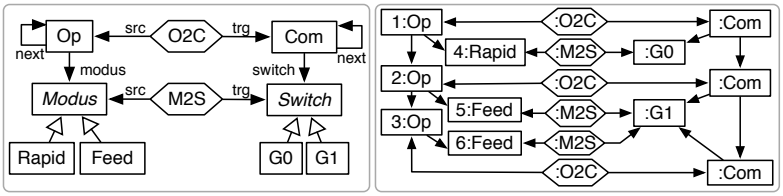


Fig. 2. A type triple graph (left), and a typed triple graph (right)

from the type triple graph. According to the type triple graph, a CLS model is represented as an ordered sequence of operations (Op), each equipped with a Modus, which can be either of type Rapid or Feed. The CLS specification in Fig. 1 is, therefore, represented by a sequence of three Ops: the first with a Rapid modus, and the subsequent two with Feed modi (the kind of CLS operation is given by the connected Mode of the Op). Similarly, a G-code model is represented by an ordered sequence of commands (Com) each equipped with a Switch either of type G0 or G1. The G-code model for the running example is a sequence of three Coms, where the first one has a G0 switch and the other two "share" a G1 switch. The sharing of the G1 switch represents the omission of unchanged information (the switch) on the last line of the G-code program (Fig. 1). Finally, the correspondence model consists of O2C and M2S links connecting related source and target elements.

As not all typed graph triples represent consistent pairs of CLS and G-code models, TGG rules are, therefore, required to further specify the actual language of meaningful CLS and corresponding G-code models. Due to the following Fact. 2, rules, derivations and graph grammars can be defined as in Def. 3, but for typed triple graphs and typed triple morphisms.

Fact 2 (The Category of Typed Triple Graphs)
The class of all typed triple graphs and typed triple morphisms form a category called **TriGraphs** *with the set $\mathcal{M}$ of injective typed triple morphisms.*

Proof. For the proof we refer the interested reader to Fact. 4.18 in [3].

Definition 7 (Triple Rules, Triple Graph Grammar).
Let TG be a type triple graph, and $L, R \in \mathcal{L}(TG)$.
A typed triple morphism $r : L \to R$ is a triple rule *if r_S, r_C, and r_T are rules.*
A triple graph grammar *$TGG = (TG, \mathcal{R})$ consists of a type triple graph TG and a finite set $\mathcal{R}$ of triple rules.*
The source-correspondence grammar *$sc(TGG)$ of a triple graph grammar $TGG = (TG, \mathcal{R})$ is a pair $(sc(TG), sc(\mathcal{R}))$, where $sc(\mathcal{R}) = \{sc(r) \mid r \in \mathcal{R}\}$ is the respective set of source-correspondence morphisms (rules).*
A conditional application condition $((a_S, a_C, a_T), \vee_{i \in I} c_i)$ is a conditional source application condition *if a_C and a_T are identities (conditional target and correspondence application conditions are defined analogously).*

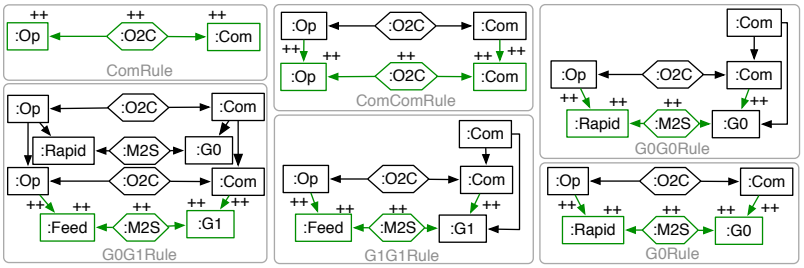


Fig. 3. Triple rules for running example

Example 2. The six TGG rules for the running example are depicted in Fig. 3. Every triple rule $r : L \to R$ is depicted in a compact syntax by denoting created elements, i.e., $R \backslash L$, with a ++ markup (and by displaying them in green). ComRule creates an Op and a Com connected by an O2C correspondence, depicted as a hexagon to improve readability. ComComRule also creates a corresponding pair of Op and Com elements, requiring a preceding triple as context to which the new elements are connected. The remaining rules handle the creation of corresponding Modi and Switches. The rules show that we have decided to allow possibly redundant repetitions of G0 switches with G0Rule, as well as reusing the G0 switch from the previous Com with G0G0Rule. This degree of freedom is, therefore, not restricted by the TGG and can be decided upon at runtime. For G1 switches, however, we have chosen to forbid redundant repetitions already at design time, i.e., a new G1 switch can only be created with G0G1Rule if it is impossible to reuse the G1 switch of the previous Com with G1G1Rule. The running example (Fig. 2) is consistent as it can be created by the following derivation: ComRule $\to$ G0Rule $\to$ ComComRule $\to$ G0G1Rule $\to$ ComComRule $\to$ G1G1Rule.

2.2 Operationalization of Triple Graph Grammars

Although TGG rules can be used directly to generate consistent source and target models, e.g., for testing purposes, TGGs are often *operationalized* to derive unidirectional forward and backward transformations [13,9]. To this end, a *forward rule* is derived from each TGG rule according to the following definition.

Definition 8 (Derivation of Forward Rules).
A TGG $= (TG, \mathcal{R})$ *without conditional application conditions can be* forward operationalized *by deriving FWD(TGG) := (TG,* $\mathcal{R}_F$*), where* $\mathcal{R}_F$ *consists of* forward rules. *A forward rule* $(r_F : FL \to FR, \mathcal{N}_F)$ *for triple rule* $r \in \mathcal{R}$ *is defined by the diagram below (the triangle denotes a set of NACs):*

$$\begin{array}{ccccccc} L & = & L_S & \xleftarrow{\sigma_L} & L_C & \xrightarrow{\tau_L} & L_T \\ {\scriptstyle r}\downarrow & & {\scriptstyle r_S}\downarrow & & {\scriptstyle r_C}\downarrow & & {\scriptstyle r_T}\downarrow \\ R & = & R_S & \xleftarrow{\sigma_R} & R_C & \xrightarrow{\tau_R} & R_T \end{array} \quad \Rightarrow \quad \begin{array}{ccccccc} & & & & \mathcal{N}_F & & \\ FL & = & R_S & \xleftarrow{r_S \circ \sigma_L} & L_C & \xrightarrow{\tau_L} & L_T \\ {\scriptstyle r_F}\downarrow & & {\scriptstyle id}\downarrow & & {\scriptstyle r_C}\downarrow & & {\scriptstyle r_T}\downarrow \\ FR & = & R_S & \xleftarrow{\sigma_R} & R_C & \xrightarrow{\tau_R} & R_T \end{array}$$

$\mathcal{N}_F$ *consists of correspondence NACs* $(id, n_C, id) : FL \to (R_S \overset{\sigma_{FN}}{\leftarrow} N_C \overset{\tau_{FN}}{\rightarrow} N_T)$ *for every node* $v_C \in V_{R_C} \setminus V_{L_C}$ *(cf. Def. 7).* N_C *and* N_T *are extensions of* L_C *and* L_T *by* v_C *and* $\tau_{R_C}(v_C)$*, respectively.*

Remark 3 (Avoiding Conflicts in Forward Rules)
For presentation purposes, we assume that TGG rules are always constructed so that every created source and target element is connected to at least one new correspondence element. With this assumption, forward rules with correspondence NACs derived according to Def. 8 "translate" every source element exactly once and simplify the formalization as compared to introducing translation attributes [8], or bookkeeping [9]. To avoid unnecessary create/forbid *conflicts,* $\mathcal{N}_F$ *can be further extended by application conditions to avoid obvious dead-ends when applying forward rules. We only give an intuition of how this works with our running example and refer to [9,8] for a description of* filter NACs.

Example 3. Figure 4 depicts the forward rules derived from the triple rules in Fig. 3, where labels indicate the original triple rules (e.g., ComFwdRule derived from ComRule). Forward rules "translate" the source elements that are created in the respective triple rules by attaching correspondence elements to them, but only if the correspondence elements do not already exist, i.e., the elements have not already been translated. Similarly, context elements of source models in triple rules are required to be already translated in forward rules by demanding an attached correspondence element. ComFwdRule, for example, "translates" an Op by connecting it to a new O2C and a Com in the target model. This should only be possible if the Op has not been translated already (hence the correspondence NAC denoted here as a crossed out (forbidden) element). An additional source NAC in ComFwdRule forbids the existence of a previous Op, i.e., this forward rule can translate only the first Op in a sequence of connected Ops. If this is not prevented, the incoming edge from a previous Op would no longer be translatable as the original triple rules cannot create such an edge between two existing Ops. This is automatically detected and prevented by generating a source "filter" NAC (cf. Remark 3).

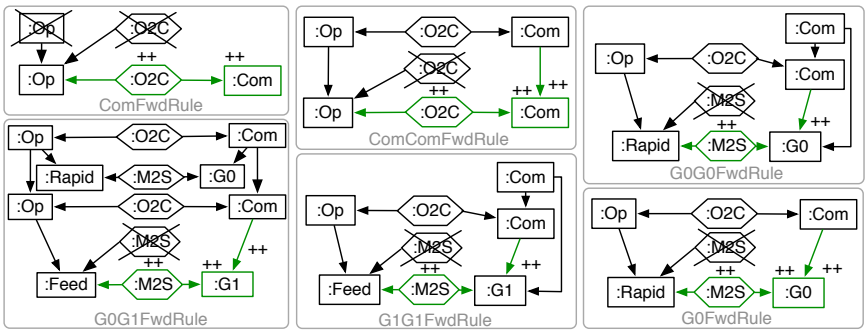


Fig. 4. Derived forward rules for the running example

The forward rule ComComFwdRule requires a translated previous Op by demanding an O2C as context, which can only exist if the connected Op has already been translated. All other forward rules are constructed analogously.

3 Efficient Model Transformation with TGGs

In addition to forward rules, a *control algorithm* is required to determine in what order source elements can be translated by forward rules. The challenge is to accomplish this translation efficiently without compromising formal properties. This has been shown in [9] for the algorithm depicted in Alg. 1. The algorithm consists of a main loop on Line 4 where a derivation with a source-correspondence rule is searched for. If none can be found, the loop is terminated and the resulting triple graph is expected to be consistent with respect to the TGG (Line 11). If this is not the case, an error is thrown on Line 12. Every derivation with a source-correspondence rule is extended to a derivation with a forward rule on Line 5. Note that this is done *locally* by extending m and not searching the whole triple graph for some suitable m' globally. If this extension is not possible, an error is thrown on Line 8. For non-confluent TGGs, the algorithm requires a choice (user or runtime configuration module) in the procedure chooseAndApplyFWDRule.

Algorithm 1 TGG-Based Forward Transformation

Require: $TGG = (TG, \mathcal{R})$ without application conditions, $\exists\, G \in \mathcal{L}(TGG)$.

1: **procedure** FwdTransform($TGG, G_S \xleftarrow{\emptyset} \emptyset \xrightarrow{\emptyset} \emptyset) : G_S \leftarrow G_C \rightarrow G_T$
2: $\quad TGG_F = (TG, \mathcal{R}_F) \leftarrow \mathrm{FWD}(TGG)$
3: $\quad G \leftarrow (G_S \xleftarrow{\emptyset} \emptyset \xrightarrow{\emptyset} \emptyset)$
4: $\quad$ **while** $\exists\, (G \overset{sc(r)@m}{\Longrightarrow} H)$ in $sc(TGG_F)$, $r : FL \rightarrow FR \in \mathcal{R}_F$ **do**
5: $\quad\quad$ **if** $\exists (G \overset{r'@m'}{\Longrightarrow} H')$ in $TGG_F, r' : FL' \rightarrow FR' \in \mathcal{R}_F :$
6: $\quad\quad\quad \exists\, e : sc(FL) \rightarrow FL' \in \mathcal{M}, m = m' \circ e$ **then**
7: $\quad\quad\quad G \leftarrow$ chooseAndApplyFwdRule(m)
8: $\quad\quad$ **else** Error: No applicable forward rule at m
9: $\quad\quad$ **end if**
10: $\quad$ **end while**
11: $\quad$ **if** $G \in \mathcal{L}(TGG)$ **then return** G
12: $\quad$ **else** Error: No applicable source-correspondence rule $sc(r)$ for G
13: $\quad$ **end if**
14: **end procedure**

In the worst case, every source element is visited exactly once in the main loop, i.e., as often as there are elements in the source model (n_S times). Each time, a source-correspondence derivation must be determined and extended. This is bounded by $|\mathcal{R}_F| \cdot n^k$, where k is the number of elements of the largest forward rule, and n is the size of the resulting triple. With n^k as the upper bound for pattern matching, the resulting complexity is $O(n_S \cdot |\mathcal{R}_F| \cdot n^k) = O(n^k)$ (cf. [9]).

Our goal in this paper is to provide a static analysis for a class of TGGs for which this algorithm never fails (as on Lines 8 and 12). To avoid non-applicability of source-correspondence rules to not fully translated graphs (error on Line 12), we have to guarantee that every derivation $G \overset{*}{\Longrightarrow} H$ in $sc(TGG_F)$ can be prolonged if $G \notin \mathcal{L}(TGG)$, i.e., not all source elements have been translated. This is a well-known property that follows from *confluence* (of $sc(TGG)$ in this case) and can be checked statically via a *critical pair* analysis (cf., e.g., [3]), i.e., with well-known techniques already applied in the context of TGGs as in [8].

Definition 9 (Confluence, Source-Correspondence Confluence).
A pair $P_1 \overset{*}{\Longleftarrow} K \overset{*}{\Longrightarrow} P_2$ *of derivations in a graph grammar is* confluent *if there exists an* X *together with derivations* $P_1 \overset{*}{\Longrightarrow} X$ *and* $P_2 \overset{*}{\Longrightarrow} X$*. A graph grammar is* confluent *if all pairs of its derivations are confluent. A triple graph grammar* TGG *is* source-correspondence confluent *if* $sc(TGG)$ *is confluent.*

To prevent dead ends (error on Line 8), it must be possible to extend every source-correspondence derivation locally to a forward derivation. The following definition formulates a TGG property that prevents the errors thrown in Alg. 1.

Definition 10 (Efficiency of Triple Graph Grammars).
Let $TGG = (TG, \mathcal{R})$ *without conditional application conditions and* $FWD(TGG) = (TG, \mathcal{R}_F)$*.* TGG *is* forward efficient *for* $G \in \mathcal{L}(TGG)$ *if:*
(1) It is source-correspondence confluent, and with $TGG_F = (TG, \mathcal{R}_F)$
(2) $\forall G' \in \mathcal{L}(TGG_F, G_S \overset{\emptyset}{\leftarrow} \emptyset \overset{\emptyset}{\rightarrow} \emptyset) : \exists (G' \overset{sc(r)@m}{\Longrightarrow} H)$ *with* $r : FL \to FR \in \mathcal{R}_F$
$\Rightarrow \exists (G' \overset{r'@m'}{\Longrightarrow} H'), r' : FL' \to FR' \in \mathcal{R}_F$ *such that* m' *extends* m*, i.e.,*
$\exists\, e : sc(FL) \to FL' \in \mathcal{M}$ *such that* $m = m' \circ e$*.*

Example 4. Considering the translation of the CLS model in Fig. 2, not all its direct derivations with source-correspondence rules can be extended locally to direct derivations with forward rules. For example, indicating the source elements translated by each application after the @ sign, the sequence sc(ComFwdRule)-@1:Op $\to_1$ sc(G0FwdRule)@4:Rapid $\to_2$ sc(ComComFwdRule)@2:Op $\to_3$ sc(ComComFwdRule)@3:Op $\to_4$ sc(G1G1FwdRule)@6:Feed $\to_5$ sc(G0G1FwdRule)@5:Feed is a derivation of source-correspondence rules that translates the CLS model. However, the fifth direct derivation sc(G1G1FwdRule)@6:Feed cannot be extended locally to a direct derivation with any forward rule. This is because the forward translation of 6:Feed before 5:Feed is not possible, although "parsing" them, i.e., the source-correspondence translation is. When translating 6:Feed, G1G1FwdRule requires a G1 switch, which can only be present if 5:Feed has already been translated. In general, this hidden dependency on 5:Feed cannot be accounted for when parsing the source model with source-correspondence rules. The TGG, therefore, does not fulfil the efficiency requirement of Def. 10 and can lead to a runtime error in Alg. 1. The formulation of Condition (2) in Def. 10, however, can only be checked at runtime. The following definition introduces the concept of a *local completeness constraint*, which can be statically checked, leading to a condition that will be shown to be sufficient for Condition (2) in Def. 10.

Definition 11 (Local Completeness Constraints, Local Completeness).
Let $TGG = (TG, \mathcal{R})$ be a triple graph grammar without conditional application conditions, $TGG_F = FWD(TGG) = (TG, \mathcal{R}_F)$.
For a forward rule $r_F : FL \to FR \in \mathcal{R}_F$, the set $lcc(r_F)$ of local completeness constraints *is defined as:*

$$lcc(r_F) := \{c \in \mathcal{M} \mid \exists\, r'_F : FL' \to FR' \in R_F, c : sc(FL) \to FL'\}$$

The local completeness constraint *$lcc(TGG_F)$ is defined as:*

$$lcc(TGG_F) := \wedge_{r_F \in R_F}(\vee_{c \in lcc(r_F)} c)$$

TGG_F is locally complete *if $TGG_F \models lcc(TGG_F)$.*

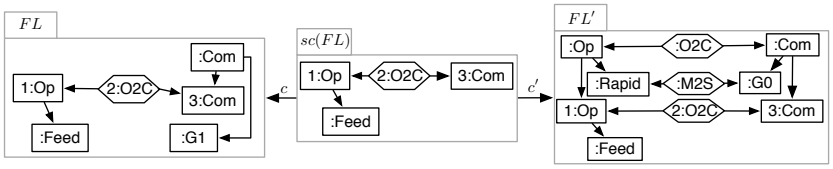


Fig. 5. Set of local completeness constraints $lcc(G1G1FwdRule)$

Example 5. Figure 5 depicts the set $\{c, c'\}$ of local completeness constraints for G1G1FwdRule, i.e., lcc(G1G1FwdRule). In the middle, the premise of the constraint is constructed as the source-correspondence graph $sc(FL)$ of the left-hand side FL of G1G1FwdRule. The two constraints result from the two possible conclusions constructed by determining all left-hand sides of forward rules (FL of G1G1FwdRule itself, and FL' of G0G1FwdRule) into which this premise can be injectively mapped. At least one of these constraints must be fulfilled, i.e., the local completeness constraint $lcc(TGG_F)$ is a disjunction of all such constraints. For the running example, the constraint demands that every occurrence of the source-correspondence context of G1G1FwdRule imply the context of at least one forward rule, i.e., in this case the context of G1G1FwdRule or of G0G1FwdRule. This means that an operation has to be already translated into a command with a preceding G0- or G1-command, *before* the feed of the operation is translated.

Local completeness constraints can be transformed to application conditions using the construction given in Fact. 1. In this manner, the forward rules of a TGG can be statically checked for local completeness by demanding that only trivial application conditions are generated. This idea is stated in the following.

Corollary 1 (Enforcing Local Completeness).
Let $TGG = (TG, \mathcal{R})$ be a triple graph grammar without conditional application conditions and $TGG_F = FWD(TGG) = (TG, \mathcal{R}_F)$. For every forward rule $r_F : FL \to FR \in \mathcal{R}_F$, there is a construction A producing application conditions $A(lcc(TGG_F), r_F)$. $(TG, \mathcal{R}'_F)$ with forward rules with the constructed application conditions $(r_F, A(lcc(TGG_F), r_F))$ is locally complete.

Proof. Follows directly from Fact. 1 as the given construction has been shown in [1] to be applicable for **TriGraphs**.

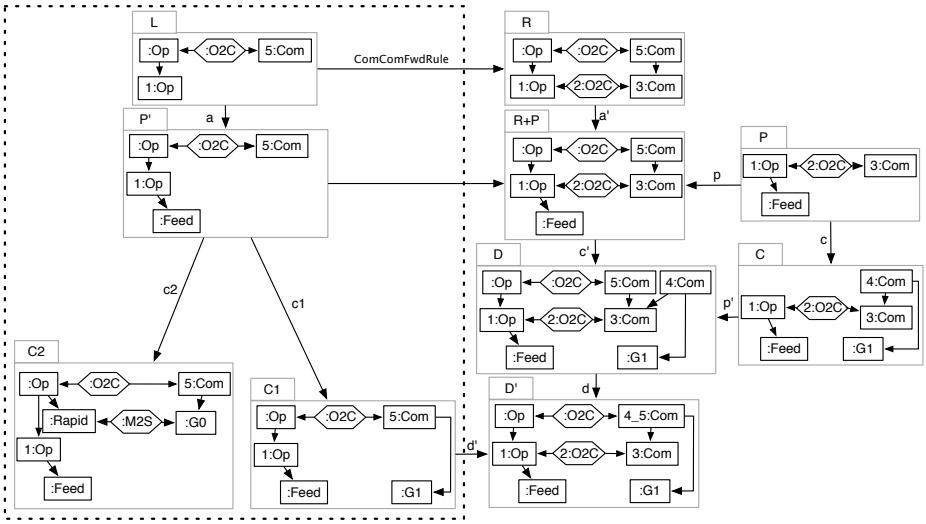


Fig. 6. Construction of local completeness application condition for ComComFwdRule

Example 6. To check if the TGG for the running example is locally complete, the application condition $(a, c_1 \vee c_2)$ depicted in Fig. 6 is constructed for the forward rule ComComFwdRule, from the local completeness constraints lcc(G1G1Fwd-Rule) (cf. Fig. 5). The steps of the construction are only shown for $c1$. As the generated application condition is not trivial, the forward rules of the TGG are not locally complete. For the running example, the application condition states that if an Op has a Feed, then it can only be translated with ComComFwdRule, if the Modus of the previous Op has already been translated. Note that $c1$ is a bit subtle, only demanding that a $G1$ be present and not caring how it is connected. The problematic sequence, obtained by translating 6:Feed with ComComFwdRule *before* 5:Feed, violates $c1$ as the Com of the previous Op does *not* yet have a G1.

We can now present our main contribution, a static analysis to check forward efficiency as defined in Def. 10 using Algorithm 2.

Theorem 1 (Static Analysis of Non-Confluent TGGs).
TGG = (TG, $\mathcal{R}$) is forward efficient if LCA(TGG) = true.

Proof. $\forall\, G \in \mathcal{L}(TGG)$, we must show forward efficiency for TGG (Def. 10): $(\mathrm{LCA}(TGG) = \mathrm{true}) \Longrightarrow TGG_F = \mathrm{FWD}(\mathrm{TGG})$ is *source-correspondence confluent* due to the check on Line 3 of Alg. 2, and is *locally complete* as the generated application conditions for local completeness are trivial (Line 5 of Alg. 2).

Let $sc(r) : SL \to SR \in sc(\mathcal{R}_F), r \in \mathcal{R}_F$, with $TGG_F = (TG, \mathcal{R}_F)$.
$\forall\, G' \in \mathcal{L}(TGG_F, G_S \xleftarrow{\emptyset} \emptyset \xrightarrow{\emptyset} \emptyset) : \exists\, (G' \overset{sc(r)@m}{\Longrightarrow} H)$
$\overset{Cor.1}{\Longrightarrow} \exists\, r' : FL' \to FR' \in \mathcal{R}_F, \exists\, c : SL \to FL' \in lcc(r) : G' \models c$
$\overset{Def.4}{\Longrightarrow} \exists\, m' : FL' \to G', m = m' \circ c \overset{Def.3}{\Longrightarrow} \exists\, (G' \overset{r'@m'}{\Longrightarrow} H')$ in TGG_F
$\overset{Def.10}{\Longrightarrow} TGG_F$ is forward efficient. □

Algorithm 2 Local Completeness Analysis (LCA)

Require: $TGG = (TG, \mathcal{R})$ without conditional application conditions.
1: **procedure** LCA(TGG) : Boolean
2: $TGG_F \leftarrow$ FWD(TGG)
3: **if** isSourceCorrConfluent(TGG_F) **then**
4: $\mathcal{AC}_{LC} \leftarrow$ constructLocalCompAppConditions(TGG_F)
5: **return** isTrivial($\mathcal{AC}_{LC}$)
6: **else return** $false$
7: **end if**
8: **end procedure**

Example 7. The TGG for the running example can be made forward efficient by stating the hidden dependency on the previous modus explicitly in ComComRule. The corrected version of this rule is depicted to the left of Fig. 7. To the right, the derived application condition $(a^*, c1^* \vee c2^*)$ is now trivial, i.e., is always fulfilled as a Switch is either G0 or G1, meaning the previous problem is solved.

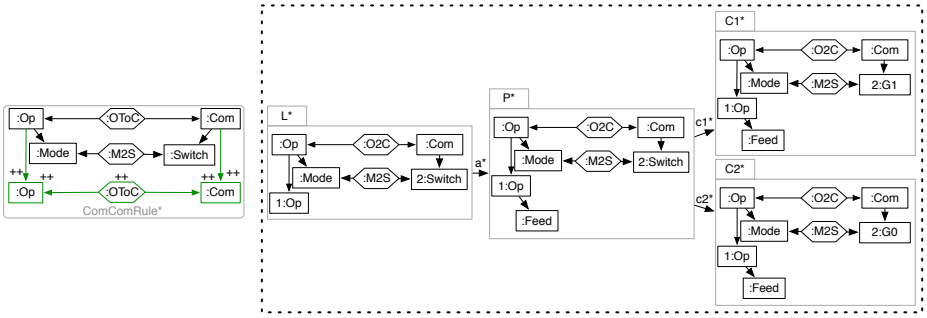


Fig. 7. Corrected TGG rule and trivial application condition for local completeness

4 Related Work

Schürr discusses the challenge of dealing with decision points in a TGG-based transformation process proposing two solutions in [13]: backtracking wrong decisions, or demanding confluence. For efficiency, the latter is the favoured strategy taken by all existing TGG approaches we are aware of. Hermann et al. [8] perform a critical pair analysis of forward rules and generate filter NACs to resolve critical pairs arising from obviously misleading (backtracking) paths. All other critical pairs (conflicts) must either be manually checked to be confluent, or removed by adjusting the TGG rules as required for confluence. More restrictively, Giese et al. [6] require confluence without filtering backtracking paths automatically. Although OCL constraints can be used to resolve critical pairs, this is a manual process required for both forward and backward transformations and it remains unclear how such OCL constraints must be restricted to guarantee formal properties. The TGG approach taken by Greenyer and Rieke [7] does not explicitly require confluence but, in case of decision points, the approach may fail in finding a valid transformation result, i.e., completeness is not guaranteed for non-confluent TGGs. Although confluence avoids backtracking (i.e., is used to show efficiency), solves completeness problems, and can be statically checked (cf. [8,6,7]), it can be too restrictive in practical scenarios (as in our running example) as it forces a TGG to be a bijection (a function in both directions).

The TGG algorithm in [9] is the only efficient and complete approach we are aware of that embraces *non-confluent* TGGs requiring local completeness and only source-correspondence confluence, as discussed in this paper. As the results in [9] do not provide any means to analyze this restriction statically, our contribution fills this gap by exploiting a constraint-based formalization of the required condition, making it amenable to well-known techniques.

Alternatives to a *static* analysis include analyses based on TGG rules *and* a concrete input model triple such as the dangling edge check in [9], and checks based on a *precedence graph* as presented in [10]. Although such analyses must be repeated for every new input model, they allow violations of properties that are not relevant for the current input model. Finally, using tools such as Groove [5] or Henshin [2], complex properties can be checked by exploring the state space generated by applying the rules of a TGG. This allows for checking arbitrarily complex conditions but suffers from the usual problem of state space explosion.

5 Conclusion

Based on our running example, taken from an industrial project in the domain of concurrent manufacturing engineering, we have argued that support for non-confluent TGGs is required for many practical scenarios. With the proposed static *local completeness analysis*, users can model degrees of freedom in consistency relations with TGGs, integrating runtime (user) interaction to decide between multiple applicable rules without compromising formal properties.

A limitation of our static analysis is that it is restricted to TGGs *without initial application conditions*. To handle constraints in the source and target

metamodels, however, such application conditions are necessary and must also be taken into account. Further tasks include providing corresponding tool support to fully automate the proposed static analysis. This requires addressing the challenge of efficiently generating application conditions from constraints, filtering out redundant results, and presenting the results in a helpful manner.

References

1. Anjorin, A., Schürr, A., Taentzer, G.: Construction of Integrity Preserving Triple Graph Grammars. In: Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.) ICGT 12. LNCS, vol. 7562, pp. 356–370. Springer (2012)
2. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 10, LNCS, vol. 6394, pp. 121–135. Springer (2010)
3. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. An EATCS Series, Springer (2006)
4. Ehrig, H., Habel, A., Ehrig, K., Pennemann, K.H.: Theory of Constraints and Application Conditions : From Graphs to High-Level Structures. Fundamenta Informaticae 74(1), 135–166 (2006)
5. Ghamarian, A.H., de Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and Analysis Using GROOVE. STTT 12 14(1), 15–40 (2012)
6. Giese, H., Hildebrandt, S., Lambers, L.: Toward Bridging the Gap Between Formal Semantics and Implementation of Triple Graph Grammars. Tech. Rep. 37, Hasso-Plattner Institute (2010)
7. Greenyer, J., Rieke, J.: Applying Advanced TGG Concepts for a Complex Transformation of Sequence Diagram Specifications to Timed Game Automata. In: Schürr, A., Varró, D., Varró, G. (eds.) AGTIVE 11. LNCS, vol. 7233, pp. 222–237. Springer (2011)
8. Hermann, F., Ehrig, H., Golas, U., Orejas, F.: Efficient Analysis and Execution of Correct and Complete Model Transformations Based on Triple Graph Grammars. In: Bézivin, J., Soley, M.R., Vallecillo, A. (eds.) MDI 10. vol. 1866277, pp. 22–31. ACM Press (2010)
9. Klar, F., Lauder, M., Königs, A., Schürr, A.: Extended Triple Graph Grammars with Efficient and Compatible Graph Translators. In: Schürr, A., Lewerentz, C., Engels, G., Schäfer, W., Westfechtel, B. (eds.) Festschrift Nagl, LNCS, vol. 5765, pp. 141 – 174. Springer (2010)
10. Lauder, M., Anjorin, A., Varró, G., Schürr, A.: Efficient Model Synchronization with Precedence Triple Graph Grammars. In: Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.) ICGT 12. LNCS, vol. 7562, pp. 401–415. Springer, Bremen, Germany (2012)
11. Leblebici, E., Anjorin, A., Schürr, A.: A Catalogue of Optimization Techniques for Triple Graph Grammars. In: Fill, H.G., Karagiannis, D., Reimer, U. (eds.) Modellierung 14. LNI, vol. 225, pp. 225–240. GI (2014)
12. Leblebici, E., Anjorin, A., Schürr, A.: Developing eMoflon with eMoflon. In: ICMT 14. LNCS, Springer (2014)
13. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 94. LNCS, vol. 903, pp. 151–163. Springer (1995)