Test Case Generation for Rule-based Translators

Martin Horsch <martin.horsch@bawue.de>

Universität Stuttgart
Fakultät Informatik, Elektrotechnik und Informationstechnik
Institut für Formale Methoden der Informatik

student research project (Studienarbeit) with internal document nr. 1984

Contents

Abstract

Graph transformation systems are a model used for the specification of programs as well as concurrent or non-deterministic systems in general. This document discusses translators that modify elements of an input language in accordance with graph rewriting rules. Such systems have in common that their state space is infinite or very large. Instead of proving their correctness, programmers and designers usually have to observe their behaviour in a finite set of test cases that can be selected automatically or by hand. An implementation of an automatic test case generator is presented, including a description of data formats used for the exchange of information on hypergraphs, graph grammars, Petri graphs and nets. The code for the generator is based on AUGUR, a set of tools for processing graph grammars. A survey on used components and modifications to the code of AUGUR is given.

1 Introduction

The development of complex devices and programs for use in sensitive areas would be unconceivable without tools for testing large systems automatically that do not require excessive calculations to be undertaken by hand. On an abstract level, we will specify such systems as non-deterministic translators starting with an element of a certain input language and modifying it step by step. It is often useful to consider graphs, instead of words, as elements of these languages, and the system itself as a set of rules describing allowed modifications of labelled graphs. This set is also called a *graph transformation system*.

There are several reasons to use labelled graphs as a model. In some cases, a program we want to test does actually use labelled graphs as its input. This is typical for automatic code generators that translate a model in a graphical notation such as *Simulink* or *Stateflow* to executable code. Another application occurs when the input language of a program is ambiguous. Most of the time, the ambiguity is resolved by the parser, which guesses an arbitrary or the most probable resolution and passes a representation of it to a component of the software that does the actual work. If that component is to be tested, the natural approach is to execute it on such internal representations instead of possibly ambiguous words of the input language. Often enough, these representations as generated by a parser are labelled graphs.

With the abstract notion of the system as a translator, one can follow an approach to selecting test cases that is in a way analogous to condition coverage: if a system is represented by a set of rules, we are interested in obtaining, for each subset of this set, maybe up to a certain size, an element of the input language from the specification of the system to which all rules of the subset can be applied. In [2], the foundations are laid out for an algorithm that takes the input language of a translator and its transformation rules as an argument and attempts to construct a test case for some of its rules. Such a test case generator was implemented as a part of this student research project (Studienarbeit). It is based on the current version of AUGUR, a set of programs for the analysis of graph transformation systems, and is called **atcg**, the AUGUR test case generator.

The document is organized as follows. Section 2 introduces graph grammars, nets and unfoldings. Section 3 describes on-the-fly and sequential firability checking as two general approaches that can be followed, as well as some modifications and restrictions that are useful to get rid of infinite state spaces. Section 4 gives a survey on **atcg** and the interaction between its internal components and external modules called by them. Section 5 explains what document formats **atcg** uses and describes them briefly, Section 6 refers to the parts of the code for AUGUR where relevant modifications were made, and Section 7 concludes the document with some remarks on software in general.

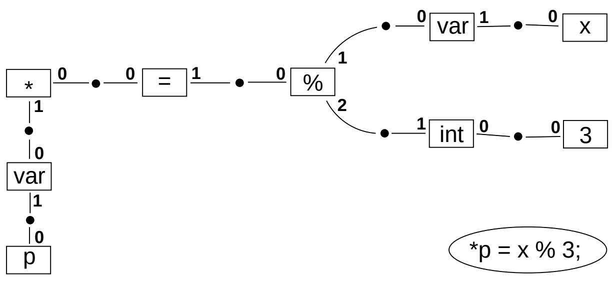

Figure 1: Hypergraph representing a line of C code

2 Graph Transformation Systems and Unfoldings

2.1 Labelled Hypergraphs

In the chosen approach to test case generation, we regard a translator as a process that, step by step, changes and replaces components of an input graph in order to produce an element of its target language. Since both input and output as well as intermediate states are represented by graphs, the topic of this document belongs to the domain of graph rewriting theory, used as a tool for analyzing dynamic systems. In [6] an elaboration of this theory is given that, together with [2], serves as the base for all of the following. More information is also provided by the Handbook of Graph Grammars [14].

Definition (Label Structure)
Let Σ be a set of labels. Then, $\Lambda \subseteq \Sigma \times \mathbf{N}$ is a *label structure* for Σ. For an element $(\lambda, ar_\lambda) \in \Lambda$, we call ar_λ an *arity value* of λ.

Definition (Hypergraph)
Let Λ be a label structure for Σ. A *directed Λ-hypergraph* is a tuple $G = (V, E, c, \lambda)$, where V is a set of vertices and E is a set of *edges* with $V \cap E = \emptyset$, $c : E \to V^*$ is a total *connection function* and $\lambda : E \to \Sigma$ a total *labelling function* that satisfies the arity condition:

$$\forall e \in E, n \in \mathbf{N} : \quad |\, c(e) \,| = n \quad \Rightarrow \quad (\lambda(e), n) \in \Lambda$$

Intersection and union of hypergraphs are defined componentwise. For a vertex or edge d we write $d \in G \Leftrightarrow d \in (V \cup E)$.

When drawing a hypergraph, rectangles are used to represent edges. Numbers can be annotated to the connections between edges and nodes to specify their order in the word $c(e) \in V^*$. Undirected hypergraphs have $c : E \to \mathbf{N}^V$, thus mapping an edge to a multi-set instead of a tuple of vertices. They are visualized just like directed hypergraphs, but without annotations of the type described above.

The arity values of a label λ indicate how many nodes are connected to an edge labelled with λ. For instance, code in a programming language can be rendered as a graph where labels are names of operators and functions. Then, arity values can correspond to the number of arguments taken by an operator. Consider Figure 1, a model of a code fragment in C. Here, we have $(\%, 3) \in \Lambda$, thus edges labelled with the modulo operator connect to three nodes: one for the return value and another two for the arguments of the operator.

For a label structure where all arity values are 2, a Λ-hypergraph corresponds to an edge-labelled (di)graph. In the following we will refer to directed Λ-hypergraphs as hypergraphs, whenever it is understood - or irrelevant - which label structure we use.

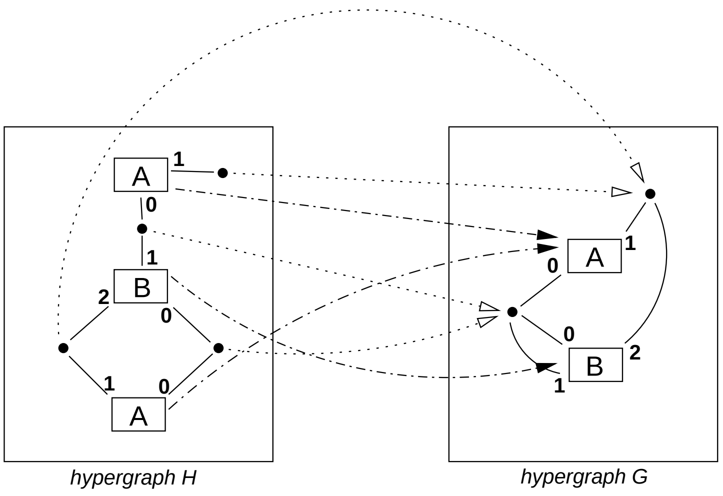

Figure 2: A surjective, non-injective hypergraph morphism φ with $\varphi(H) = G$

2.2 Graph Transformation Systems

We will use *graph transformation systems* (GTS) to describe the way a translator alters its input. Graph transformation systems are analogous to the sets of production rules that occur in phrase-structure grammars, differing from them mainly in the aspect that they operate on graphs instead of words.

Definition (Subgraph)
Let Λ be a label structure and $G = (V, E, c, \lambda)$, $H = (V_H, E_H, c_H, \lambda_H)$ be Λ–hypergraphs. H is called a *subgraph* of G ($H \subseteq G$) iff $V_H \subseteq V$, $E_H \subseteq E$ and

$$\forall e \in E_H : \quad c_H(e) = c(e) \wedge \lambda_H(e) = \lambda(e).$$

If, in addition to this, every vertex $v \in V_H$ is connected to at least one edge $e \in E_H$, we call H the subgraph of G generated by E_H.

Definition (Hypergraph Morphism)
Let Λ be a label structure and $G = (V, E, c, \lambda)$, $G' = (V', E', c', \lambda')$ be Λ-hypergraphs. A hypergraph morphism $\varphi : G \to G'$ is a pair of functions $(\varphi_V : V \to V', \varphi_E : E \to E')$, such that the following property holds, where $\varphi_V^* : V^* \to (V')^*$ is the homomorphism generated by φ_V.

$$\forall e \in E : \qquad \lambda(e) = \lambda'(\varphi_E(e)) \quad \wedge \quad \varphi_V^*(c(e)) = c'(\varphi_E(e))$$

A hypergraph morphism $\varphi = (\varphi_V, \varphi_E)$ with injective or surjective components φ_V *and* φ_E is also called injective or surjective, respectively. For a surjective morphism $\varphi : H \to G$ (compare Figure 2) we write $\varphi(H) = G$. An injective morphism $\varphi : H \to G$ is called a *match* of H in G, a bijective morphism is called an *isomorphism*. Two hypergraphs G and H for which an isomorphism exists are *isomorphic*, denoted as $G \cong H$. A match $\varphi = (\varphi_V, \varphi_E)$ of H in G indicates that there is a subgraph $\varphi(H) \subseteq G$ isomorphic to H.

Definition (Graph Rewriting Rule)
A graph rewriting rule is a tuple $r = (I, L, R, \varphi_L, \varphi_R, \nabla)$. I, L and R are hypergraphs, where I stands for the *context*, L and R for the *left* and the *right hand side* of the rule; φ_L and φ_R are matches of the context in L and R, respectively, and ∇ is a set of *negative application conditions* (NAC). The hypergraph L is generated by its edges, i.e. it does not contain any isolated vertices, and each condition $\nu \in \nabla$ is a match of L in a hypergraph $\nu(L) + e_\nu$ that exceeds $\nu(L)$ by exactly one edge e_ν, the *inhibitor edge* of ν.

A set of rewriting rules - all for the same label structure Λ - is also called a Λ-*graph transformation system* (GTS).

Definition (Enabled Match)
Let $r = (I, L, R, \varphi_L, \varphi_R, \nabla)$ be a rewriting rule and G a hypergraph in which there is a match $\psi = (\psi_V, \psi_E)$ of L that can not be extended to include an inhibitor edge of r. That is, for no $\nu \in \nabla$, there is a match $\psi' = (\psi'_V, \psi'_E)$ of $\nu(L) + e_\nu$ in G such that, for all edges e in $\nu(L)$, $\psi_E(\nu^{-1}(e)) = \psi'_E(e)$ holds. Then, ψ is an *enabled match* of r in G and the rule r is *enabled* in G by the means of ψ.

Where and whenever r is enabled in G, a *rewriting step* can occur that replaces its left hand side in G by its right hand side, just like applying a production rule in a phrase-structure grammar does.

Definition (Rewriting Step)
Let $L = (V_L, E_L, c_L, \lambda_L)$ and $G = (V_G, E_G, c_G, \lambda_G)$ be hypergraphs, $\varphi_L = (\varphi_{V_L}, \varphi_{E_L})$ and $\varphi_R = (\varphi_{V_R}, \varphi_{E_R})$ morphisms, $r = (I, L, R, \varphi_L, \varphi_R, \nabla)$ a rule and $\psi = (\psi_V, \psi_E)$ an enabled match of r in G. The rewriting step of r applied to G according to ψ is a pair of hypergraphs (G, H) with $H = (V_H, E_H, c_H, \lambda_H)$, also denoted as $G \Rightarrow_r H$, for which there is a match $\psi' = (\psi'_V, \psi'_E)$ of R in H and the following assertions hold:

1. R accounts for all additions to G.
   $H \subseteq G \cup \psi'(R)$
2. Context is preserved, edges of L not mirrored in I are destroyed.
   $\forall e \in E_L : \quad \psi_E(e) \in H \Leftrightarrow e \in \varphi_L(I)$
3. ψ and ψ' are in agreement, they describe the same section of the graph.
   $\forall e \in I : \quad \psi_E \circ \varphi_{E_L}(e) = \psi'_E \circ \varphi_{E_R}(e)$
4. Preserved edges are connected to the same vertices in G and in H.
   $\forall e \in E_G \cap E_H : \quad c_G(e) = c_H(e)$

A *rewriting sequence* $s = r_1 r_2 \cdots r_n$ of rules r_i is enabled by G iff there is a sequence of hypergraphs $(H_1, H_2, \ldots H_n)$ such that $G \Rightarrow_{r_1} H$ and $\forall 1 < i \leq n : H_{i-1} \Rightarrow_{r_i} H_i$. This is symbolized by $G \Rightarrow^*_s H_n$.

We say that G and H are isomorphic *up to isolated vertices* - in symbols: $G \simeq H$ - iff the graphs generated by their edges are isomorphic. If $G \simeq H$, we can substitute G by H without enabling or disabling any rewriting sequences, since the left hand side of a rule must not contain any isolated nodes.

Definition (Graph Grammar)
Let Λ be a label structure. A Λ-*graph grammar* is a pair $\mathfrak{G} = (P, S)$, where P is a Λ-graph transformation system and S is a Λ-hypergraph called the *start graph*. The union of $\mathfrak{G}$ and a GTS P' is a graph grammar again: $\mathfrak{G} \cup P' = (P \cup P', S)$.

In a graphical representation, the left and right hand sides of a rewriting rule are displayed as hypergraphs with barred inhibitor edges e_ν attached to the left hand side. The matches φ_L and φ_R are reflected by ascribing identical numbers to corresponding items.

Consider Figure 3, taken from [2]. It shows a grammar that generates arithmetic expressions. In this example, numbers are annotated to the edges labelled with $+$. Such attributes can be interpreted *1)* as a higher level notation supported by higher level graph grammars only, *2)* as an abbreviation for additional edges that contain these attributes as labels - as is done in Section 4.7 - or *3)* as a part of the label (which multiplies the number of rules in the grammar).

The language produced by a graph grammar could, for example, be defined in analogy with phrase-structure grammars by declaring some of the labels as non-terminal and the others as

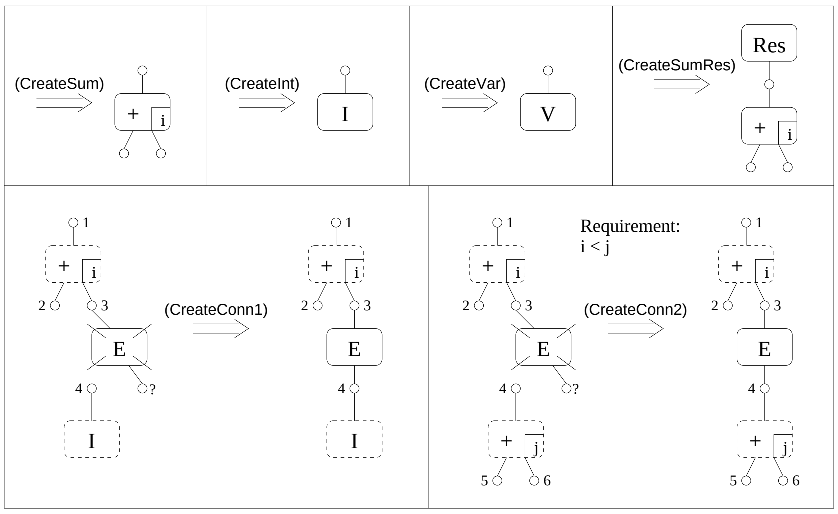

Figure 3: A graph grammar that generates arithmetic expressions

terminal. In most cases, however, graph grammars are not used with the intention of reasoning about the languages they generate. Instead, we are more interested in knowing what rules can be applied in which order, how they depend on each other and what intermediate states can occur. In a certain sense, we will consider all labels as terminal.

Definition (Test Case)
Let $\mathfrak{G}_0 = (S, P_0)$ be a graph grammar called the *generator*, P_1 be a GTS called the *translator* and $\mathfrak{G} = \mathfrak{G}_0 \cup P_1$ have the set $P = P_0 \cup P_1$ of production rules. Let $P_T \subseteq P_1$ be a *target* subset of the translator. For $S \Rightarrow_s^* C$ with $s \in P_0^*$, we call C a *test case* of P_T iff it enables a rewriting sequence $s' \in P_1^*$ in which every element of P_T occurs at least once.

An example for a translator is given in Figure 4; it contains simplification rules for arithmetic expressions from the generating grammar in figure 3. In this example, the translator is therefore an epxression optimizer. Figure 13 in Section 4 shows a test case for the rule *KillUselessFunction*.

2.3 Context-dependent Place/Transition Nets

Petri nets are often used to represent networks and concurrent processes. They were introduced by Carl Adam Petri in his dissertation [11], which was the starting point of what is today a large segment of theoretical computer science. This document can only mirror parts of this theory that are immediately relevant for the task of finding test cases for a GTS. An introduction to Petri net theory can be found in [12] or [13].

We will follow [2] in considering a generalized form of Petri Nets that allows *read* and *inhibitor* arcs, which reflect the behaviour of graph grammars in a natural way. For such nets, we use the notion of "context-dependent nets" with positive and negative "context relations" that Montanari and Rossi introduced in order to mirror the effects contexts and negative application conditions have in graph rewriting theory [9].

A place/transition net is essentially an automaton with a set of *places*, each of which can contain a number of *tokens*, with *transitions* operating on them. When a transition is *fired*, i.e. executed, the number of tokens in its neighbouring places changes depending on the *arcs* that connect them.

Definition (Context-dependent Place/Transition Net)
Let S be a set of places and T a set of transitions with $S \cap T = \emptyset$ and let $b, f, \rho : T \times S \to \mathbf{N}$

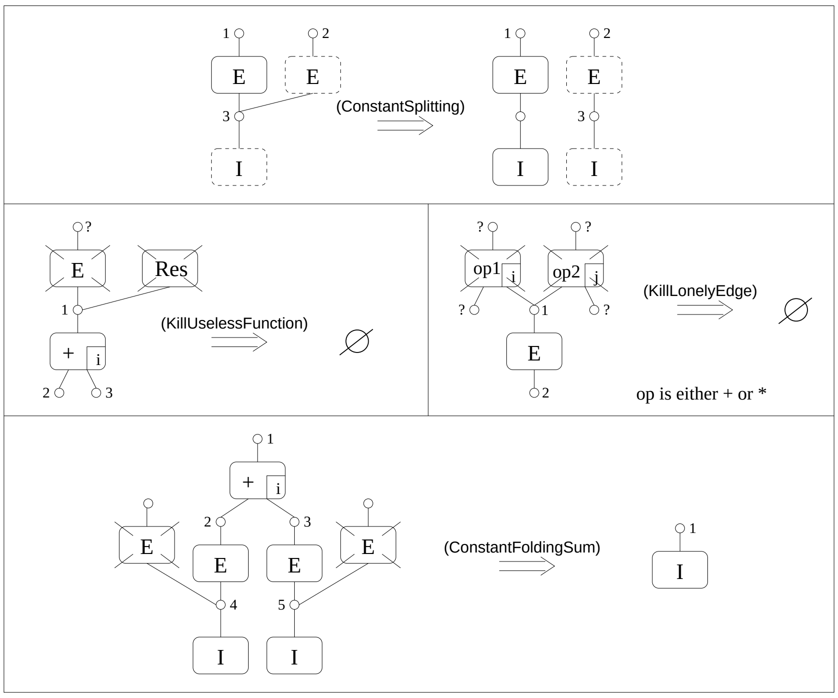

Figure 4: Simplification rules for arithmetic expressions (compare [2])

and $\chi : T \times S \to \mathbf{N}^+ \cup \{\infty\}$ be total functions. Then, the tuple $N = (S, T, b, f, \rho, \chi)$ is a *context-dependent place/transition net* (CPN) and we call b, f, ρ and χ the *backward*, *forward*, *read* and *inhibitor* functions of N. For $s \in S, t \in T$ the following sets and arcs are defined:

1. ${}^\bullet t = \{s \in S \mid b(t, s) > 0\}$ is the *preset* of t. (by analogy: ${}^\bullet s = \{t \in T \mid f(t, s) > 0\}$)
   For $s \in {}^\bullet t$, a (backward) write arc of weight $b(t, s)$ is leading from s to t.

2. $t^\bullet = \{s \in S \mid f(t, s) > 0\}$ is the *postset* of t. (by analogy: $s^\bullet = \{t \in T \mid b(t, s) > 0\}$)
   For $s \in t^\bullet$, a (forward) write arc of weight $f(t, s)$ is leading from t to s.

3. $\underline{t} = \{s \in S \mid \rho(t, s) > 0\}$ is the *context set* of t.
   For $s \in \underline{t}$, s and t are connected by an undirected *read arc* of weight $\rho(t, s)$.

4. $\multimap t = \{s \in S \mid \chi(t, s) \neq \infty\}$ contains the *inhibitor places* of t.
   For $s \in (\multimap t)$ - also: $s \multimap t$ - an *inhibitor arc* of weight $\chi(t, s)$ is leading from s to t.

For $b(t, s) = f(t, s)$ the resulting pair of arcs is also called a *write-back arc*. A function $M : S \to \mathbf{N}$ is a *marking* of N. We say that, for a given marking M, a place $s \in S$ contains $M(s)$ tokens, and that s is *marked* iff $M(s) > 0$.

Definition (Firing of Transitions)
Let $N = (S, T, b, f, \rho, \chi)$ be a CPN and M a marking of N. A transition $t \in T$ is *enabled* by M whenever

$$\forall s \in S : \; b(t, s) + \rho(t, s) \leq M(s) < \chi(t, s).$$

Iff t is enabled by M, it can be *fired*, an event upon which t *replaces* M by the marking

$$M' : S \to \mathbf{N}, \; s \mapsto M(s) + f(t, s) - b(t, s),$$

in symbolic notation: $M \to_t M'$.

Intuitively, firing a transition t involves the following steps: **1)** for each inhibitor arc with weight χ_s from a place s to t, **assert** that s contains less than χ_s tokens. **2)** for each write

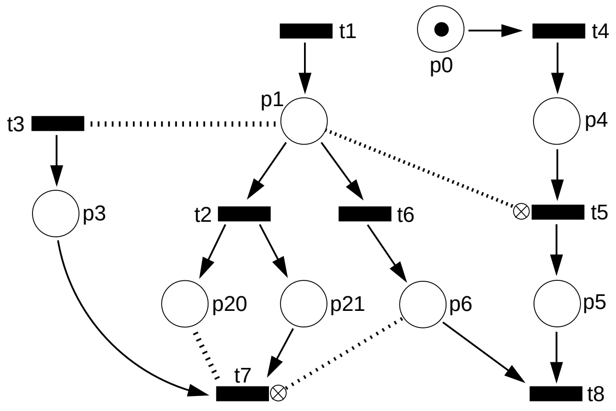

Figure 5: A context-dependent place/transition net

arc with weight b_s from a place s to a place t, **assert** that s contains at least b_s tokens; remove b_s tokens from s. **3)** for each read arc with weight ρ_s between t and a place s, **assert** that s contains at least ρ_s tokens. **4)** for each write arc with weight f_s from t to a place s, add f_s tokens to s. **5)** commit the effects of the transition if all assertions have been confirmed.

In a graphical representation, transitions are visualized as filled rectangles, places as large circles and tokens as small filled circles. We draw write arcs as arrows, read arcs as undirected dashed lines and inhibitor arcs as dashed lines with a crossed circle at the end (an example is given in Figure 5). As long as nothing different is said, we suppose that all arcs in a CPN have weight 1, i.e. that b, f, ρ and χ are functions from $T \times S$ to $\{0, 1\}$.

Typically, such a diagram does not only show a P/T net, but also a distribution of tokens over the net. In most cases, the marking indicated that way is regarded as the *initial marking* of the net, and the attention is directed to the transitions that are enabled at this marking and to possible future markings of the net.

Definition (Reachable Marking)
Firing sequences and the transitive closure $\rightarrow^*$ of $\rightarrow$ are defined for CPN the same way as rewriting sequences and the relation $\Rightarrow^*$ were defined for graph grammars in Section 2.2. Let N be a CPN and M_0 its initial marking. A function M is called a *reachable marking* of N if and only if it is a marking of N and there is a firing sequence $\sigma \in T^*$ enabled in M_0 such that $M_0 \rightarrow^*_\sigma M$.

Definition (Safe Nets, Occurrence Nets and Petri Nets)
For $k \in \mathbf{N}$, a marking M with $\forall s \in S : M(s) \leq k$ is *k-safe* and, if $k = 1$, simply *safe*. A CPN with initial marking is called (k-)safe, iff all of its reachable markings are (k-)safe. A CPN without cycles of write arcs is *acyclic*, and a safe acyclic CPN with with $\forall s \in S : {}^\bullet s \leq 1$ is also called an *occurrence net*. A CPN without read and inhibitor arcs is a *Petri Net*.

Without loss of generality, the initial marking of an acyclic (1-)safe CPN has the property that exactly the places with an empty preset contain a token, while the others are unmarked. In a k-safe CPN, we can assume without loss of generality that

$$s \multimap t \implies \chi(t, s) - 1 \leq k + b(t, s) - f(t, s)$$

for all places s and transitions t (otherwise the inhibitor arc is redundant) and that all write, read, and inhibitor arcs have a weight lower or equal to k. Safe CPN are a special case of context-dependent condition/event nets as developed in [9]. A safe CPN can be transformed info a safe Petri net efficiently without affecting its reachable markings and enabled firing sequences (compare Section 3.3).

Definition (Dependency between Transitions)
Let N be an occurrence net and S its set of places. Let T be the transition set of N. The *causality* relation $(\cdot < \cdot) \subseteq T \times T$ is the least transitive relation such that

$$\forall t, \tau \in T: \quad t^\bullet \cap ({}^\bullet\tau \cup \underline{\tau}) \neq \emptyset \;\Rightarrow\; t < \tau$$

If $t \in \tau$, this means that τ is *causally dependent* on t in the sense that in every enabled firing sequence of N the transition τ can only occur after t if at all, because t is the only transition that can mark the preset or the context of τ. The set of transitions on which τ depends is also called the *cause* $\lfloor\tau\rfloor$ of τ. When dependency between transitions is visualized, we draw a minimal generating subset of $<$ instead of the whole relation, because additional arcs make the representation harder to understand and are just redundant (the causality relation for the example CPN is contained as a part of Figures 9 and 12).

Definition (Conflict between Transitions)
Let t and τ be two transitions of the CPN N. We say that t and τ are in (symmetric) conflict $t\#\tau$ iff ${}^\bullet t \cap {}^\bullet\tau \neq \emptyset$. For $t\#\tau \;\vee\; \underline{t} \cup {}^\bullet\tau \neq \emptyset$ the *asymmetric conflict* $t \nearrow \tau$ is defined.

Two transitions with common elements in their preset are in conflict, because in a safe marking that enables both, firing one of the transitions disables the other one. If t and τ are in asymmetric conflict, no safe marking enables the sequence τt, i.e. they must be fired in the order indicated by the relation $\nearrow$ or with additional transitions inserted in between to restore the tokens. In an occurrence CPN *without inhibitor arcs*, a set T' of transitions can be fired if and only if the asymmetric conflict relation has no cycles of elements of its cause $\lfloor T'\rfloor = \cup_{t\in T'}\lfloor t\rfloor$.

Finding a possible firing order of transitions in an occurrence CPN with inhibitor arcs corresponds to the task of finding a *configuration*, which is defined formally in [5] and applied to this question in [1] and [2]. For our purposes, a configuration $(\cdot <_{\mathbf{C}} \cdot) \subseteq T \times T$ of a CPN $N = (S, T, b, f, \rho, \chi)$ is a relation such that $(\cdot < \cdot) \cup (\cdot <_{\mathbf{C}} \cdot)$ is acyclic and for all $s \multimap \tau$, where $'s$ is (the only) element of ${}^\bullet s$:

$$\tau <_{\mathbf{C}} {}'s \quad \vee \quad \exists t \in s^\bullet : \; t <_{\mathbf{C}} \tau$$

If s has an empty preset, it is initially marked and only the second option is allowed, which corresponds to the intuition that τ can only be fired after the token has been removed from s. The difficulty of that problem is due to the fact that even in an occurrence net we can not tell in advance if for $s \multimap \tau$, τ should be fired before s is filled or after s is emptied.

3 Infinite Unfoldings and Finite Prefixes

3.1 Net and Graph Grammar Unfoldings

Causality and conflict can be applied much easier to occurrence nets than to CPN in general, so in order to use these properties for analyzing a CPN, we are interested in obtaining an occurrence net that is in some sense equivalent to it. This is called an *unfolding*, a concept introduced by McMillan in [8].

An unfolding N' of a CPN N is an occurrence net that enables a set of firing sequences congruent with the set of firing sequences allowed by N. This is achieved by copying transitions of N, possibly infinitely often, and the congruence of firing sequences is defined by the means of the surjective function from the transitions of N' to the transitions of N that maps each copy to the corresponding original. An algorithm for unfolding nets is presented in [5].

The notion of unfoldings can be extended to graph grammars. Unfolding a graph grammar $\mathfrak{G}$ yields a *Petri graph* $P = (G, N)$ where G is a hypergraph and N is a context-dependent occurrence net. The edges of G are identical with the places of N, and the initial marking of N corresponds

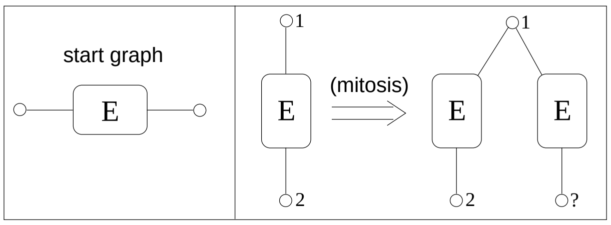

Figure 6: A simple graph grammar

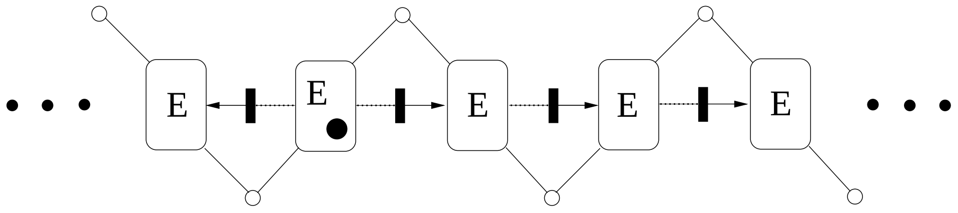

Figure 7: Unfolding of the graph grammar from Figure 6

to the start graph of G. The transition set of N is mapped to the ruleset of $\mathfrak{G}$ such that the firing sequences enabled by the initial marking of N are congruent with the rewriting sequences enabled by the start graph of $\mathfrak{G}$. The marking of P is the subgraph of G generated by the marked edges, i.e. the places marked in N. The N component of the Petri graph can also be transformed to a safe Petri net as described below. Compare [6] for further information on unfolding graph grammars.

In a visualization, we display the CPN and the hypergraph together, in a way such that edges are also used as places and can contain tokens. Consider the graph grammar of Figure 6. Figure 7 shows a part of its unfolding. It is generated inductively by beginning with the start graph of $\mathfrak{G}$ and attaching an instance of a rule, i.e. a transition that corresponds to it, to every match (enabled or not) of its left hand side that is found in the graph. Backward write arcs correspond to the edges deleted by an instance of the of the rule, forward write arcs to the edges it creates, read arcs to its context and inhibitor arcs to the matches of left hand sides extended by inhibitor edges $\nu(L) + e_\nu$ in the graph.

In almost all cases, the unfolding of a graph grammar is infinite, which makes it impossible to operate on the Petri graph as a whole. This is, first of all, due to the fact that translators are usually built for infinite languages and the unfolding of the generator-translator-system includes the entire input language of the translator. One rule similar to that shown in Figure 6 is enough to generate a Petri graph of infinite size.

In the few cases of translators for finite languages, correctness can be verified by having them translate a complete dictionary or, if that is impossible, which means that their input language is still very large, the unfoldings are even larger and can not be processed either. Therefore, some restrictions are necessary to allow for an algorithmic analysis of their behaviour.

3.2 Unfolding Only as Far as Necessary

For finding a test case, a finite prefix of the unfolding, i.e. a connected subgraph that includes the start graph, is sufficient, supposed that the start graph actually enables a rewriting sequence that includes each of the target rules. However, we do not know now in advance how large the unfolding must be for our purpose, and in general, there is no way to decide whether a test case can be found or not (a graph grammar can emulate a Turing machine). A test case generator must therefore constrain the size of the unfolding and eventually give up if it can not find a result.

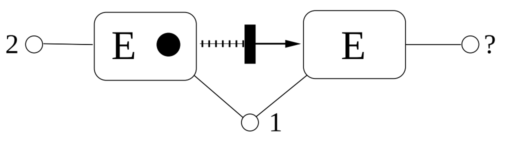

Figure 8: Petri graph for the *mitosis* rule from Figure 6

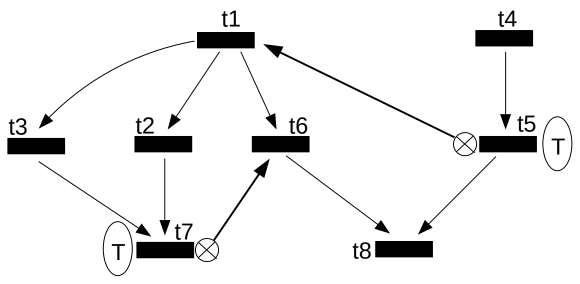

Figure 9: Causality relation with a possible configuration for firing t_5 and t_7

Restrictions on the size of the Petri graph alone are insufficient, because a finite net can still have an infinite state space. Consider the Petri graph of Figure 8 that corresponds to the *mitosis* rule applied to the start graph, where arbitrarily many tokens can be fired into the right edge. This problem can be overcome by adding a *dummy place*, intially marked with one token, to the preset of every transition that does not reduce the marking of any other place [2]. In the resulting Petri graph, there is no transition that can be fired more than once. Since in an unfolding, every place s has $| {}^\bullet s | \leq 1$, we obtain a *safe* Petri graph.

A test case generator can unfold a grammar step by step according to breadth first search or some kind of heuristics and every time an instance of a target rule is inserted to the Petri graph, search for a configuration that allows this transition and at least one instance of all other target rules to be fired. This way of attacking the problem can be referred to as *on-the-fly firability checking*, because there are interleaving steps of grammar unfolding and attempts to find configurations for firing transitions.

When there are few interactions between the restrictions imposed by inhibitor arcs, finding a configuration or proving it impossible is easy, as in the case of the net from Figure 5, where $<_C$ can be constructed independently for the inhibitor arcs leading to t_7 and t_5. One of the two possible solutions is displayed in Figure 9.

In an appendix to [2], some ideas are presented concerning the search for configurations in more complicated cases. For the implementation, a different approach was chosen, which will be described in the following section.

3.3 Modular Approach to Unfolding and Coverability Checking

Instead of scanning for test cases on the fly while the unfolding of the grammar is constructed, it is also possible to unfold the grammar first and the CPN obtained as a part of the resulting Petri graph afterwards. Although such a modular approach is, in principle, by no means more efficient than a monolithic one (rather the other way round), it allows the exploitation of existing software for the two major parts of the problem: **1)** to construct a finite prefix of the unfolding, and **2)** to detect a firing sequence that contains instances of all target rules. This approach can be referred to as *sequential firability checking* because it consists of two distinct steps.

The first step of the task can be delegated to an external GTS unfolder. For this purpose, the unfolding must be limited a priori. In [2], two kinds of restrictions are proposed. They have to be combined in order to get a finite occurrence graph:

- **depth restriction k:**
  a rule r is unfolded and a transition t added to the graph, if in the extended graph, no transitions $t_1, \dots, t_k$ would exist such that $t_1 < \cdots < t_k < t$, where $<$ is the causality relation over the CPN component of the Petri graph.

- **width restriction w:** every instance of the rule is unfolded exactly w times.

In our case **aunfold**, a part of AUGUR, is used to create an unfolding prefix of width 1 and variable depth. NAC are not processed and must be evaluated separately (compare Section 4).

For the second step, the Model Checking Kit can be invoked to check coverability properties of safe Petri nets [15]. Before this is possible, the CPN must be transformed into a Petri net, which is accomplished by introducing a *complement place* for every place of the original net. This is possible because in the original net all places contain, for all reachable markings, either exactly one token or none. In general, complementation as described below works for all k-safe nets. At this stage of the problem, we can be sure the CPN is even (1-)safe, supposed that dummy places with an initial marking of 1 have been added to all transitions with empty presets.

Construction of a Petri net equivalent to a safe CPN
Let $N = (S, T, b, f, \rho, \chi)$ be a k-safe CPN with functions $b, f, \rho, \chi : T \times S \to \{0, \dots, k\}$ and the initial marking M_0. Let S' consist of all $s \in S$ and a new place $\bar{s}$ for every s. We define $\forall s \in S, t \in T$: $M'_0(\bar{s}) = k - M_0(s)$ and:

- $b'(t,s) = b(t,s) + \rho(t,s) \;\wedge\; f'(t,s) = f(t,s) + \rho(t,s)$
- $s \multimap t \;\Rightarrow\; b'(t,\bar{s}) = k - \chi(t,s) + 1 \;\wedge\; f'(t,\bar{s}) = b(t,s) - f(t,s) + k - \chi(t,s) + 1$
- $s \not\multimap t \;\Rightarrow\; b'(t,\bar{s}) = f(t,s) \;\wedge\; f'(t,\bar{s}) = b(t,s)$

We call $N' = (S', T, b', f', 0, 0)$ the Petri net equivalent to N, where 0 is the constant zero function, and M'_0 its initial marking. The invariant $M(s) + M(\bar{s}) = k$ holds by induction for all reachable markings M of N' and all $s \in S$, since for all transitions $t \in T$:

$$(f'(t,s) - b'(t,s)) \;+\; (f'(t,\bar{s}) - b'(t,\bar{s})) \;=\; 0$$

In the modified net, for a transition t and $s \multimap t$
the firing condition is $b'(t,s) \leq M(s) \wedge b'(t,\bar{s}) \leq M(\bar{s})$ or

$$b(t,s) + \rho(t,s) \leq M(s) \quad \wedge \quad k - \chi(t,s) < k - M(s),$$

which corresponds to the firing condition for N. Firing a transition affects the marking of s equally in both nets, which means that N and N' enable the same firing sequences.

Actually, it is sufficient to complement those places that have outgoing inhibitor arcs. Look at Figure 10, which shows the Petri net equivalent to the CPN in Figure 5, to see how complementation works for (1-)safe nets: read arcs replaced by *write-back arcs* and complementary places added for *p1* and *p6*. Finally, inhibitor arcs are substituted with write-back arcs to the complementary places. In addition to this, dummy places were inserted to the presets of *t1* and *t3* (the original net from Figure 5 is not safe).

The Petri net obtained that way is equivalent to the original net in the sense that both of them enable the same firing sequences, and that a firing sequence applied to both graphs leads to equal markings of all places that are defined in both nets. In [9], a difference is pointed out: transitions that are not in conflict in the context-dependent net can be in conflict in the modified net. This difference need not be considered for our application, but can be relevant in other cases.

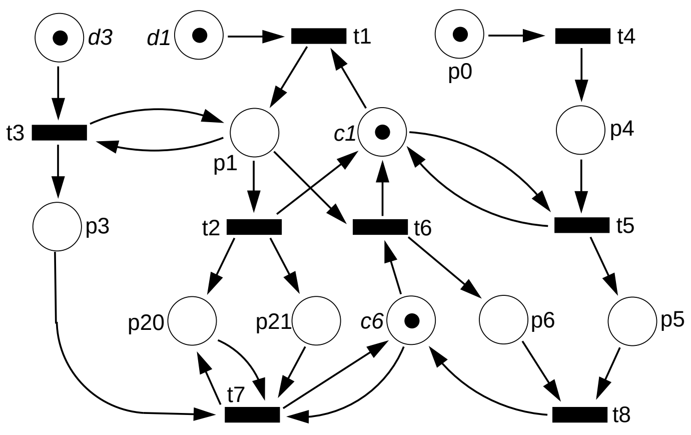

Figure 10: Safe Petri net corresponding to the P/T net from Figure 5

Now, a test case can be composed. For this purpose, we introduce an additional *target place* for each target rule and add it to the postset of every instance of the rule. For the resulting net, a coverability checker is asked for a firing sequence σ that covers the target places, that is $M_0 \rightarrow^*_\sigma M$ such that for all target places $s_r : M(s_r) \geq 1$. If such a σ is found, a solution to our problem can often be obtained by firing the first, generating part of the sequence. The subgraph generated by the marking at this point enables a sequence of transitions that includes instances of all target rules, which makes it a valid test case.

However, it may happen that for $\sigma = t_1 t_2 \cdots t_n$ there are $0 < i < j \leq n$ such that t_i corresponds to a translating and t_j to a generating rule. In such a case one can try to *serialize* the sequence by rearranging some of the transitions, which does not always work and may be impossible even if a valid firing sequence really exists (compare Section 4.7).

4 Implementing a Test Case Generator

4.1 The atcg Executable

Together with this document, the implementation of a test case generator for translators represented by GTS is released. It is based on code for the GTS analyzer AUGUR and therefore called **atcg**, the AUGUR test case generator. The executable file takes three arguments:

- **gts**: union of generating grammar and translating GTS
- **depth**: depth of the generated prefix of the unfolding
- **target**: a set of target rewriting rules

In addition to this, some options can be passed to the program:

-a	directory containing AUGUR binaries
-c	directory containing check executable (Model Checking Kit)
-d	do not unfold the graph transformation system
-e	directory containing neato executable
-f	fire up to level (normally 0)
-g	output file for the unfolding (petri graph in GXL format)
-h	display help
-n	output file for the unfolding (PEP low level net format)
-o	output file for the test case (GXL format)

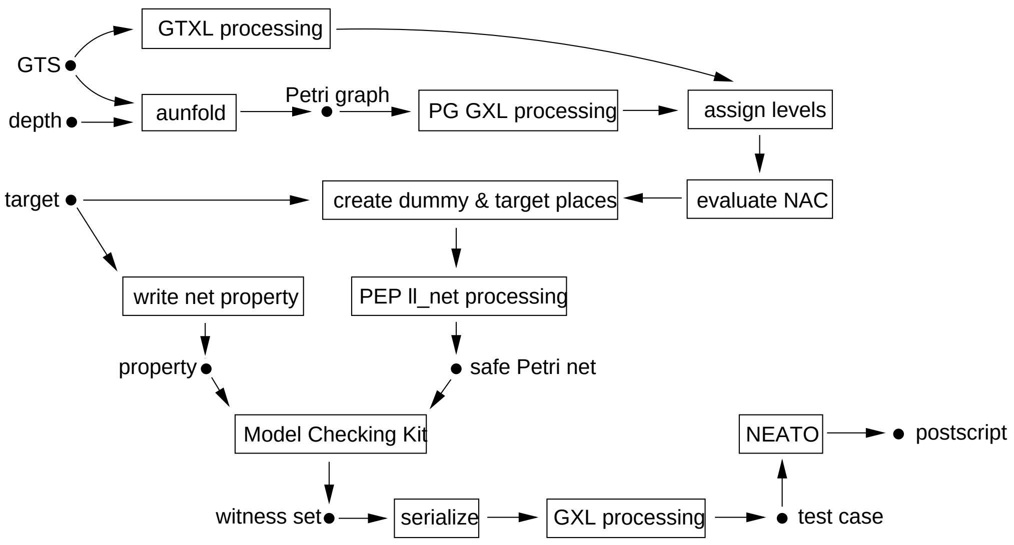

Figure 11: Program flow of the implemented test case generator

-p	output file for a visualization as postscript
-v	promulgate version number
-w	working directory (must have writing permission)

Note that **atcg** calls AUGUR binaries instead of invoking them by function calls. This allows a user to switch between different versions of AUGUR or to replace it by an equivalent set of executable files. The **aunfold** binary is however required to support the **-cc** option, which increases its performance eminently, but is missing in older versions.

If option **-d** is selected, the program will assume that *the unfolding is already up to date*. By default, if *gts.xml* is the file containing the GTS, the unfolding is loaded from *gts.unf.xml*. A different location can be indicated with option **-g**.

The rules in *gts* have attributes called their *levels* (compare Sections 4.5 and 5.3). We are interested only in executions where rules are applied in increasing order of levels. The value $\digamma$ passed together with option **-f** indicates the level where the translator begins. Rules with levels lower than $\digamma$ are considered part of the generating grammar, whereas rules with levels greater or equal to $\digamma$ are part of the translating GTS.

4.2 Combining Internal and External Modules

A modular structure combining both the execution of code that belongs to **atcg** and calls to external binaries was chosen. A survey on their interaction is given in Figure 11.

Given its three mandatory arguments *gts*, *depth* and *target*, **atcg** tries to find an element - a hypergraph - in the source language of the translator that allows, among others, the *target* rules to be applied during the process of translation. To achieve this, it performs essentially the following steps:

1. produce an unfolding of the *gts*, ignoring NAC, with a *depth* given by the user

2. assign levels to transitions of the Petri graph, i.e. the unfolding of *gts*

3. add dummy places and *target* places to the Petri graph

4. scan for matches of NAC and create complement places with write-back arcs

5. check whether the *target* places are coverable
    - **no?** abort
    - **yes?** load a firing sequence (witness set) that covers the places
6. serialize the firing sequence (abort if that is impossible)
7. build the test case by firing the generating part of the sequence

The internal and external modules implementing these steps will be described in the following passages. Compare [7] for a manual to AUGUR and its components and Section 6 of this document for a survey on the modifications and additions made to the code of AUGUR in order to obtain internal components of the test case generator.

4.3 Document Processing

Several data formats must be processed:

- Graph Exchange Language *GXL 1.0.1* (read and write)
- the variant of GXL used for Petri graphs (read and write)
- Graph Transformation Exchange Language *GTXL* (read)
- the *PEP ll_net* format for low level Petri nets (write)

For a description of these document types read Section 5 or, respectively, the literature indicated there. The library *libxml2* is used for parsing and writing the XML formats (all except for the PEP ll_net format).

4.4 Unfolding the Grammar with aunfold

The tool **aunfold** is a part of AUGUR that can construct the k-depth approximated unfolding of a graph grammar [7]. Its input is a GTXL file and its output a Petri graph in GXL format (compare Section 5). The call made to **aunfold** is:

```
aunfold -cc gts.xml gts.unf.xml
```

The width restriction from Section 3.3 is not supported directly. If all rules are to be unfolded w times for the same match, this can be achieved by placing an additional hyperedge with a special label in their context and creating w edges with that label in the start graph. NAC are ignored by **aunfold** and processed later on by a method of **atgc**.

4.5 Processing the Unfolded Grammar

Some modifications are necessary before the net produced by **aunfold** as a part of the grammar unfolding can be passed on to the model checker for safe Petri nets. We begin by adding *level* values to transitions.

In general, the level of a transition is a level of the corresponding rule. An additional requirement is raised by the fact that we want to be able to *serialize* firing sequences of the net, i.e. it should be possible to fire the transitions of level **0** first, then the transitions of level **1**, and so on. Speaking in terms of [2], we want to ensure that the model checker returns a sequence "compatible with the grammar ordering", which is a necessary condition for obtaining a test case.

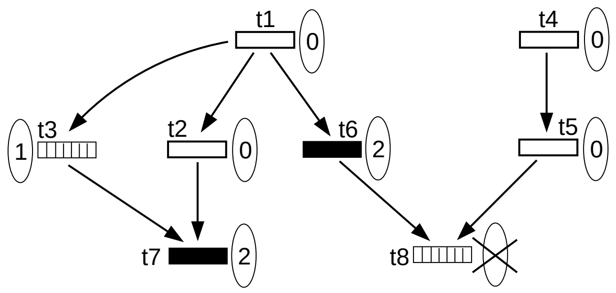

Figure 12: Levels and causality relation for the CPN from Figure 5

For that purpose we demand that the level of a transition t is larger than all levels of transitions in $\lfloor t \rfloor$. Instances of rules that operate on various levels are always given the least value compatible with this requirement. If that is impossible, the transition is removed together with the whole subgraph of the Petri graph that depends on it.

Suppose that in Figure 12, the transitions t_1, t_2, t_4, and t_5 have level **0**, t_3 and t_8 have level **1** and the others have level **2**. Then, in the net passed on to the model checker, transition t_8 will no longer appear. For the implementation, a simple DFS can be used. However, even after the removal of such transitions, the net may enable non-serializable firing sequences due to restrictions imposed by inhibitor arcs.

The next step creates additional places. As described in section 3, every transition with an empty preset is connected to a dummy place in order to guarantee that the net is safe. With this modification we know that the P/T net component of the unfolding is an occurrence net. For each target rule, an additional target place is inserted to the net, and every time we find an instance of that rule in the net, we add that place to its postset. (Although the target places can have an in-degree greater than one and be marked with more than one token, this does not turn out to be a problem.)

As **aunfold** ignores negative application conditions, **atcg** processes them in an additional step. For every rule $r = (I, L, R, \varphi_L, \varphi_R, \nabla)$ and for all $\nu \in \nabla$, all matches of $\nu(L) + e_\nu$ in the Petri graph are extended by an inhibitor arc leading from e_ν to the instance of r. The CPN is tacitly stored as the equivalent Petri net (read arcs are represented by write-back arcs and inhibitor arcs by write-back arcs to a complementary place).

4.6 Model Checking Kit for Safe Petri Nets

The Model Checking Kit is a collection of programs for modelling finite state systems that are internally represented as safe Petri nets [15]. Among many other formats, it can read a Petri net in PEP format (Section 5.4) and check whether some of its places can be covered. The call made to its binary file **check** is the following:

```
check pep:mcs-reach gts.unf.ll_net formula | tail -n 2 > witness_set
```

The *formula* is a conjunction over the target places in plain text format. The last two lines of the output are stored in a special file. If the places can be covered, they contain what is called the *witness set*, i.e. a firing sequence that covers them.

4.7 Building the Test Case

In Figure 13, you can see a rendering of the test case generated for the target rule *KillUseless-Function*. The rules of the generating grammar, i.e. those that construct the operator *Op*, have

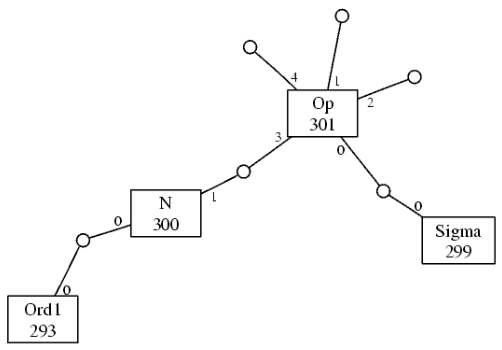

Figure 13: Test case enabling the target rule **KillUselessFunction**

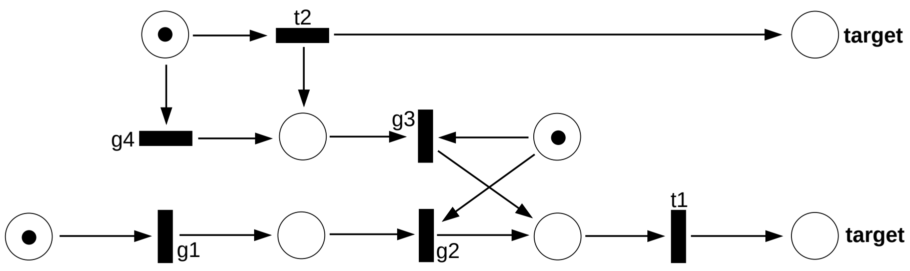

Figure 14: Net with non-serializable firing sequence $\{t_2, g_3, t_1\}$

been applied, while the target rule that belongs to the translator has not. In this case, it is no problem to serialize the witness set, because it is clearly possible to construct a useless function without applying any rules from the arithmetic expression optimizer.

Note that here, the test case does not correspond exactly to the version of the rules given in Section 2.2 and in [2]. Attributes, such as type and order of operators, the latter of them necessary to avoid cycles, had to be represented by edges with special labels. The technical reason for this is that AUGUR does not evaluate any higher level notation, but requires plain, basic GTS as defined above. For Figure 13 this means that *Op* represents an operator, *Sigma* its type (addition) and *Ord1* describes it as *first order*, i.e. not requiring the evaluation of any other operator.

If the witness set returned by the model checker does not consist of transitions with monotonously increasing levels, **atcg** trys to rearrange them accordingly. However, such an attempt need not be successful.

Consider the safe Petri net of Figure 14 and imagine that transitions labeled with g belong to the generator and have level **0**, whereas those labeled with t are instances of translating rules and have level **1**. Such a net can actually occur at this stage, because in the corresponding CPN (if we imagine that places with an in-degree greater than one are complementary places) no transition of level 0 depends on a transition of level 1.

Suppose we want to find a firing sequence for t_1 and t_2. A valid solution would be $g_1 g_2 t_1 t_2$. However, a model checker will probably return the shorter firing sequence $t_2 g_3 t_1$. This sequence is enabled by the initial marking of the net and covers the target places, but can not be serialized. The correct solution to the problem is missed (solutions that the program fails to detect for this reason occur, however, far less frequently than solutions missed due to depth and width restrictions).

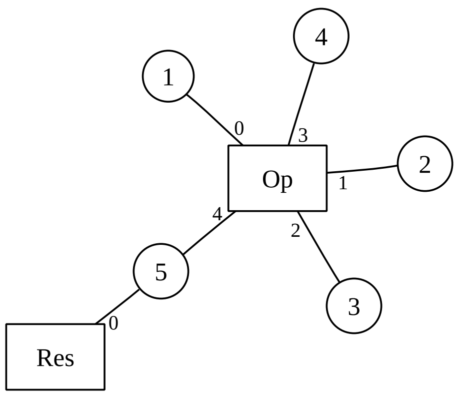

Figure 15: Example output produced by NEATO

4.8 Drawing Graphs with NEATO

NEATO is a program that layouts graphs ”by constructing a virtual physical model and running an iterative solver to find a low-energy configuration” [10]. A graph intended to be drawn by NEATO must be given in the *dot format* like the following example:

```
graph "out"
{
   _2 [shape=circle, width=.15, label="1"]
   _3 [shape=circle, width=.15, label="2"]
   _4 [shape=circle, width=.15, label="3"]
   _5 [shape=circle, width=.15, label="4"]
   _6 [shape=circle, width=.15, label="5"]
   _8 [label="Op", shape=box]
   _8 -- _2[taillabel="0", labelangle=-35, labeldistance=1]
   _8 -- _3[taillabel="1", labelangle=-35, labeldistance=1]
   _8 -- _4[taillabel="2", labelangle=-35, labeldistance=1]
   _8 -- _5[taillabel="3", labelangle=-35, labeldistance=1]
   _8 -- _6[taillabel="4", labelangle=-35, labeldistance=1]
   _9 [label="Res", shape=box]
   _9 -- _6[taillabel="0", labelangle=-35, labeldistance=1]
}
```

The result can be seen in Figure 15. Writing a hypergraph in *dot* format is currently handled by exporting it in GXL as if it were a Petri graph and then calling the **pg2neato** binary included in AUGUR.

5 Exchange Formats for Nets, Graphs and Grammars

5.1 GXL as Document Type for Hypergraphs

The *Graph Exchange Language* GXL is an XML document type used for graphs and hypergraphs. Throughout AUGUR and also in **atcg**, the GXL DTD verion 1.0.1 is used [7], which was developed by Andy Schuerr and others in 2002. An earlier version of GXL is also described in [16]. The following example corresponds to the graph from Figure 15.

```
<?xml version="1.0"?>
<gxl><graph hypergraph="true" edgemode="undirected">
```

```
   <node id="1"/><node id="2"/><node id="3"/><node id="4"/><node id="5"/>
   <rel><attr name="label"><string>Op</string></attr>
      <relend target="1" startorder="0"/>
      <relend target="2" startorder="1"/>
      <relend target="3" startorder="2"/>
      <relend target="4" startorder="3"/>
      <relend target="5" startorder="4"/>
   </rel>
   <rel><attr name="label"><string>Res</string></attr>
      <relend target="5" startorder="0"/>
   </rel>
</graph></gxl>
```

Most of this is straightforward: vertices are represented by *node* tags, hyperedges by *rel* and connections between them by *relend* tags. Labels are given by embedding a *string* content into an *attr* tag.

However, it may be confusing that the example says *edgemode="undirected"*, although the hypergraph it represents is clearly directed, as can be seen from the *startorder* values. These indicate the real order of connections between a hyperedge and the nodes attached to it. Hypergraphs must be described as undirected in any case, possibly because some programs would otherwise try to orient the connections. An undirected hypergraph is distinguished from a directed one by leaving the *startorder* value away.

For normal graphs, write *hypergraph="false"* and use

```
<edge id="..." from="..." to="...">  ...  </edge>
```

instead of *rel* tags.

5.2 Application of the GXL Format to Petri graphs

Petri graphs combine hyperedges and vertices from hypergraphs with places and transitions from Petri nets. AUGUR applies the GXL format to Petri graphs as well, treating places as hyperedges and transitions as vertices. While places and edges coincide by definition, a special notation is introduced in [7] to distinguish transitions and places.

In this variant of GXL, vertices and hyperedges get additional attributes. In particular, all edges are assigned an initial marking, which is 1 if they are part of the start graph and 0 otherwise. Arcs of the Petri net are also annotated to the hyperedges by the means of additional connections that lead to transitions. The resulting elements look as in the following example:

```
<!-- vertex -->
<node id="_126"><attr name="vertex"/></node>

<!-- hyperedge -->
<rel id="_141"><attr name="label"><string>Sigma</string></attr>
   <attr name="initial_marking"><int>0</int></attr>
   <relend target="_123" role="vertex" startorder="0"/>
   <relend target="_144" role="postset">
      <attr name="weight"><int>1</int></attr>
   </relend>
   <relend target="_164" role="preset">
      <attr name="weight"><int>1</int></attr>
```

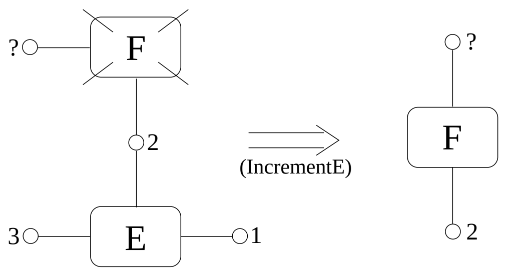

Figure 16: Rewriting rule with NAC

```
   </relend>
</rel>

<!-- transition -->
<node id="_197"><attr name="transition"/>
   <attr name="rule"><string>KillUselessFunction</string></attr>
</node>
```

5.3 Graph Transformation Exchange Language

An XML document type for graph grammars - rather than GTS, because start graphs are included - is the *Graph Transformation Exchange Language* GTXL, which is also the format **aunfold** and other executables require as input.

As the version of GTXL supported by AUGUR does not support negative application conditions, a modified document type definition had to be used for **atcg**. Essentially, the proposal from [16] for specifying NAC was followed and inserted into the version supported by AUGUR, in a way such that AUGUR binaries ignore the conditions and process the rest correctly.

A rewriting *Rule* consists of a *LHS*, a *RHS* and a *Mapping* between from vertices of the left to those of the right side. Both sides of the rule are given by a *RuleGraph* element, the left one of which can include one or more *GraphCondition* elements representing the negative application conditions. A *GraphCondition* includes another *RuleGraph* with exactly one hyperedge, the inhibitor edge, and a *Mapping* from nodes in the left hand side to the nodes connected to the inhibitor edge. The rule with inhibitor from Figure 16 looks as follows in GTXL:

```
<Rule id="IncrementE">
   <LHS><RuleGraph>
      <Graph hypergraph="true" edgemode="undirected">
         <node id="1"/><node id="2"/><node id="3"/>
         <rel><attr name="label"><string>E</string></attr>
            <relend target="1"/><relend target="2"/><relend target="3"/>
         </rel>
      </Graph>
      <GraphCondition>
         <RuleGraph>
            <Graph hypergraph="true" edgemode="undirected">
               <node id="2"/> <node id="4"/>
               <rel><attr name="label"><string>F</string></attr>
                  <relend target="2"/><relend target="4"/>
               </rel>
```

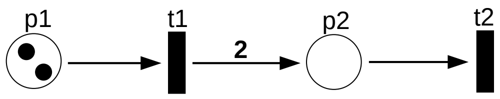

Figure 17: Petri net with a weighted arc

```
            </Graph>
         </RuleGraph>
         <Mapping><MapElem from="2" to="2"/></Mapping>
      </GraphCondition>
   </RuleGraph></LHS>
   <RHS><RuleGraph>
      <Graph hypergraph="true" edgemode="undirected">
         <node id="2"/><node id="5"/>
         <rel><attr name="label"><string>F</string></attr>
            <relend target="2"/><relend target="4"/>
         </rel>
      </Graph>
   </RuleGraph></RHS>
   <Mapping><MapElem from="2" to="2"/></Mapping>
</Rule>
```

Except for the *GraphCondition* element, this is also understood by **aunfold**. For the task of test case selection, we need an additional element to specify whether a rule belongs to the generating grammar or to the translating GTS, or more generally, for a generator with level 0 and n sequentially executed translators with levels from 1 to n, in what levels a rule can be applied. For this purpose one or more *attr* tags with embedded *int* values must be provided:

```
<Rule id="RULEID">
   <attr name="level"><int>0</int></attr>
   <attr name="level"><int>2</int></attr> ...
</Rule>
```

This attribute is ignored by other programs as well.

The AUGUR executable **rules2LaTeX** creates LaTeX input (.tex) out of a graph grammar given in GTXL - ignoring negative application conditions.

5.4 PEP Format for Low Level Petri Nets

PEP *low level nets* are a simple format for Petri nets introduced by the authors of the PEP tool [4]. It is human readable and easy to process. Look at the net from example 17 in PEP format:

```
PEP
PTNet
FORMAT_N
%
PL      % list of places
1"p1"M2 % place p1 initially marked with 2 tokens
2"p2"   % place p2 initially unmarked
%
TR      % list of transitions
1"t1"
```

```
2"t2"
%
TP      % list of arcs leading from transitions to places
1<2w2   % arc with weight 2 leading from t1 to p2
PT      % list of arcs leading from places to transitions
1>1     % arc from p1 to t1
2>2     % arc from p2 to t2
```

Instead of the PEP format one can also use the XML document type for Petri nets, the Petri Net Modelling Language (PNML), that was recently integrated into the Model Checking Kit [3].

6 Modification and Extension of AUGUR

This section indicates parts of the AUGUR code that were modified in a relevant way and describes new code that was added for **atcg**, just in case anyone wants to speed it up or reduce its memory requirements (which is actually necessary if you want to apply it to non-trivial input).

atgc.cpp
This file contains the main function that evaluates the options (compare Section 4.1) and controls the program flow (compare Figure 11 in Section 4).

class GraphGrammar
The element *string target_formula* was added. It stores the formula that is written to the property file and passed to the Model Checking Kit. Four methods were added, all of them doing what you would expect: *bool hasRule(IDTYPE id)*, *int findRule(string ruleName, bool * found)*, *void setTarget(char* target)* and *string getTargetFormula()*.

TransformationRule* GTXLReader::readRule(xmlNodePtr node)
The reader for GTXL files was adapted to the new document requirements (compare Section 5.3). In particular, NAC and levels are processed.

class Hypergraph
The class was modified in a way that allows it to add an edge to a *Hypergraph* before all of the vertices connected to it have been added, which may make sense if, in a large input file, edges are defined first and vertices later. Two elements, *bool acknowledged* and *set<IDTYPE> missing_vertices* were introduced to mirror the current state of the object. When the graph is actually used, it checks itself internally, which can also be done from the outside by calling the method *bool Hypergraph::acknowledge()*. The new method *IDTYPE extendByFirstEdgeOf(Hypergraph& h)* is used to add the inhibitor edge of a NAC to the left hand side of a rewriting rule.

class Petrigraph
Three elements were added: *map<IDTYPE, IDTYPE> gamma_mapping* stores for each hyperedge in the *Petrigraph* the id of its generating transition, *map<IDTYPE, unsigned> lambda_mapping* stores transition levels and *set<IDTYPE> removed_transitions* is used to mark transitions that are no longer considered part of the Petri graph. Various public methods were added, among them:

- *void applyLevels()* assigns levels (compare Section 4.5),
- *void applyNAC()* evaluates NAC (compare Section 4.5),
- *Hypergraph getSubgraph(PTMarking m)* returns the subgraph generated by the edges marked in m,
- *void removeSubgraph(IDTYPE transition)* removes a transition and the parts of the *Petrigraph* that depend on it, which is needed by *applyLevels()*,

- and *void writeFormula(char* filename)* writes a file with a property corresponding to the target rules (compare Section 4.6).

class PTNet

Two elements were introduced: *map<IDTYPE, IDTYPE> complement*, mapping each place with outgoing inhibitor arcs to its complementary place, and *map<IDTYPE, unsigned> capacity*, where the capacity of these places is stored, i.e. the constant sum of the marking of a place and that of its complementary place. The methods *void addReadArc(IDTYPE t, IDTYPE p, int w), addInhibitorArc(IDTYPE t, IDTYPE p, int w)* and several methods for dealing with capacities were inserted. Method *void write_llnet(char* filename)* exports the net in PEP format (Section 5.4), *PTFiringSequence readSequence(char* filename, ...)* processes a file that contains a witness set, and *PTMarking fireInAnyOrder(PTFiringSequence& ptfs, bool* isFirable)* attempts to serialize a firing sequence.

class TransformationRule

The class was changed to support negative application conditions and levels, corresponding elements were introduced. An additional flag *is_target* marks whether a rule belongs to the target set or not. Ten public methods were added, among them *IDTYPE addNAC(Hypergraph singleEdgeGraph), bool isApplicable(unsigned current)* and *unsigned getNextLevel(unsigned current)*, where the parameter *current* stands for the current operating level during the depth first search mentioned in Section 4.5.

7 Conclusion

The problem of finding test cases for subsets of a graph transformation system has been presented and an implementation of a test case generator in accordance with proposals made in [2] has been described. The followed approach was to call an external program for unfolding a graph grammar without negative application conditions, to scan the Petri graph and to add inhibitor arcs wherever matches of rules extended by inhibitor edges are found. It was shown how the net component of the unfolding is converted to a safe Petri net, which can be checked by external software as well.

Whoever tries to apply **atcg** to non-trivial examples will realize that it takes a lot of time and, what is more of a problem, is very memory-consuming. For large examples, this is clearly due to combinatorial explosion, which can not be overcome in general when graph grammars are unfolded. The problem of telling whether a rule of a graph grammar can ever be applied is undecidable, and this apparent disadvantage is at the same time the reason why systems as complex as code optimizers can even be described by this model.

Another difficulty is inherent to the chosen approach: if an application as the one presented here involves calls to external binaries, and files on the hard disk are used for communication between some of the modules, it is hard to avoid that the memory requirements multiply. In addition to this, several modules that belong to different pieces of software have to process the same information, which is slow and memory-consuming if the unfoldings are large.

In many cases, heuristics may also be a better choice than unfolding a graph grammar up to width and depth values that are constant throughout the Petri graph. On the other hand, the systematic and modular approach does also carry important benefits. Single modules can be replaced by different versions in accordance with the needs of the user, and the whole development process is easier and requires less communication. Future efforts could be directed towards a compromise between these paradigms.

Acknowledgements

I would like to thank all the people who contributed to the development of AUGUR, and especially Barbara König for proposing the topic and supervising the student research project.

References

[1] P. Baldan, N. Busi, et al. Functorial concurrent semantics for Petri nets with read and inhibitor arcs. In *Proc. CONCUR'00*, pages 442–457. Springer Verlag, 2000. LNCS 1877.

[2] P. Baldan, B. König, and I. Stürmer. Generating test cases for code generators by unfolding graph transformation systems. In *Proc. ICGT'04*, pages 194–209. Springer Verlag, 2004. LNCS 3256.

[3] J. Bart. Integration von PNML in das Model Checking Kit. Studienarbeit Nr. 1940 (student research project), Universität Stuttgart, Institut für Formale Methoden der Informatik, October 2004.

[4] E. Best and B. Grahlmann. *PEP Documentation and User Guide Version 1.8*. Universität Oldenburg, 1998.

[5] J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. In T. Margaria and B. Steffen, editors, *Proc. TACAS'96*, pages 87–106. Springer-Verlag, 1996. LNCS 1055.

[6] B. König. *Analysis and Verification of Systems with Dynamically Evolving Structures*. Habilitation thesis, Universität Stuttgart, Institut für Formale Methoden der Informatik, December 2004.

[7] V. Kozioura, B. König, et al. *AUGUR - A tool for the analysis of graph transformation systems using approximative unfolding techniques*. Universität Stuttgart, Institut für Formale Methoden der Informatik, November 2004.

[8] K. L. McMillan. A technique of state space search based on unfolding. *Formal Methods in System Design*, 6(1):45–65, 1995.

[9] U. Montanari and F. Rossi. Contextual nets. *Acta Informatica*, 32(6):545–596, 1995.

[10] S. C. North. Drawing graphs with NEATO. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, April 2002.

[11] C. A. Petri. *Kommunikation mit Automaten*. Dissertation, Universität Bonn, Rheinisch Westfälisches Institut für Instrumentelle Mathematik, 1962.

[12] L. Priese and H. Wimmel. *Theoretische Informatik: Petri-Netze*. Springer Verlag, Heidelberg, 2003.

[13] W. Reisig. *Petri Nets: An Introduction*, volume 4 of *Monographs in Theoretical Computer Science. An EATCS Series*. Springer, 1985.

[14] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations*, volume 1. World Scientific, 1997.

[15] C. Schröter, S. Schwoon, and J. Esparza. The Model-Checking Kit. In W. van der Aalst and E. Best, editors, *Applications and Theory of Petri Nets 2003*, pages 463–472. Springer Verlag, 2003. LNCS 2679.

[16] G. Taentzer. Towards common exchange formats for graphs and graph transformation systems. *Electronic Notes in Theoretical Computer Science*, 44(4), 2001.