Generating Test Cases for Code Generators by Unfolding Graph Transformation Systems*

Paolo Baldan[1], Barbara König[2], and Ingo Stürmer[3]

[1] Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy
[2] Institut für Formale Methoden der Informatik, Universität Stuttgart, Germany
[3] Research and Technology, DaimlerChrysler Berlin, Germany

baldan@dsi.unive.it koenigba@fmi.uni-stuttgart.de
Ingo.Stuermer@daimlerchrysler.com

Abstract. Code generators are widely used in the development of embedded software to automatically generate executable code from graphical specifications. However, at present, code generators are not as mature as classical compilers and they need to be extensively tested. This paper proposes a technique for systematically deriving suitable test cases for code generators, involving the interaction of chosen sets of rules. This is done by formalising the behaviour of a code generator by means of graph transformation rules and exploiting unfolding-based techniques. Since the representation of code generators in terms of graph grammars typically makes use of rules with negative application conditions, the unfolding approach is extended to deal with this feature.

1 Introduction

The development of embedded software has become increasingly complex and abstraction appears to be the only viable means of dealing with this complexity. For instance, in the automotive sector, the way embedded software is developed has changed in that executable models are used at all stages of development, from the first design phase up to implementation (*model-based development*). Such models are designed with popular and well-established graphical modelling languages such as *Simulink* or *Stateflow* from *The MathWorks*[4]. While in the past the models were implemented manually by the programmers, some recent approaches allow the automatic generation of efficient code directly from the software model via *code generators*. However, at present, they are not as mature as tried and tested C or ADA compilers and their output must be checked with almost the same expensive effort as for manually written code.

One of the main problems in code generator testing is the methodical inability to describe, in a clear and formal way, the mode of operation of the code

* Research partially supported by EU FET-GC Project IST-2001-32747 AGILE, the EC RTN 2-2001-00346 SegraVis, DFG projects SANDS and the IMMOS project funded by the German Federal Ministry of Education and Research (project ref. 01ISC31D)

[4] See www.mathworks.com

generator's transformation rules and the interaction between such rules (this is especially true for optimisation rules), a fact which makes it hard to devise effective and meaningful tests. Therefore, an essential prerequisite for testing a code generator is the choice of a formal specification language which describes the code generator's mode of action in a clear way [6].

When dealing with a code generator which translates a graphical source language into a textual target language (e.g. C or Ada), a natural approach consists in representing the generator via a set of graph transformation rules. Besides providing a clear and understandable description of the transformation process, as suggested in [19], this formal specification technique can be used for test case derivation, allowing the specific and thorough testing of individual transformation rules as well as of their interactions. We remark that, while each rule is specified with a single transformation step in mind, it might be quite difficult to gain a clear understanding of how different rules are implemented and how they can interfere over an input graph. Testing all input models triggering any possible application sequence is impractical (if not impossible), because of the large (possibly infinite) number of combinatorial possibilities, and also unnecessary as not all combinations will lead to useful results. It is, however, of crucial importance to select those test cases which are likely to reveal errors in the code generator's implementation.

In this paper we will use unfoldings of graph transformation systems [18, 4] in order to produce a compact description of the behaviour of code generators, which can then be used to systematically derive suitable test cases, involving the interaction of chosen sets of (optimising) rules. Our proposal is based on the definition of two graph grammars: the *generating grammar*, which generates all possible input models for the code generator (Simulink models, in this paper) and the *optimising grammar*, which formalises specific transformation steps within the code generator (here we focus only on optimisations). The structure obtained by unfolding the two grammars describes the behaviour of the code generator on all possible input models. Since the full unfolding is, in general, infinite, the procedure is terminated by unfolding the grammars up to a finite causal depth which can be chosen by the user. Finally, we will show how the unfolded structure can be used to select test cases (i.e., code generator input models), which are likely to uncover an erroneous implementation of the optimisation techniques (as specified within the second graph grammar). The task of identifying sets of rules whose interaction could be problematic and should thus be tested, might require input from the tester. However once such sets are singled out, the proposed technique makes it possible to automatically determine corresponding test cases, namely input models triggering the desired behaviours, straight from the structure produced via the the unfolding procedure.

The behaviour of code generators is naturally represented by graph grammars with negative application conditions [9], while the unfolding approach has been developed only for "basic" double- or single-pushout graph grammars [18, 4]. Hence, a side contribution of the paper is also the generalisation of the unfolding

construction to a class of graph grammars with negative application conditions, of which, due to space limitations, we will provide only an informal account.

The rest of the paper is structured as follows. Section 2 gives an overview of the automatic code generation approach and discusses code generator testing techniques. Section 3 presents the class of graph transformation systems used in the paper. Section 4 discusses the idea of specifying a code generator and its possible input models by means of graph transformation rules. Section 5 presents an unfolding-based technique for constructing a compact description of the behaviour of a code generator and Section 6 shows how suitable test cases can be extracted from such a description. Finally, Section 7 draws some conclusions.

2 Automatic Code Generation

In the process of automatic code generation, a graphical model, consisting for instance of dataflow graphs or state charts, is translated into a textual language. First, a working graph free of layout information is created and, in the next step, a gradual conversion of the working graph into a syntax tree takes place. In the individual transformation phases from the working graph to the syntax tree, optimisations are applied in which, for instance, subgraphs are merged, discarded or redrawn. Finally, actual code generation is performed, during which the syntax tree is translated into linear code.

In practice, a complete test in this framework is impossible due to the large or even infinite number of possible input situations. Accordingly, the essential task during testing is the determination of suitable (i.e. error-sensitive) test cases, which ultimately determines the scope and quality of the test.

In the field of compiler testing much research has been done concerning test case design, namely test case generation techniques. We can distinguish two main approaches: *automatic test case generation* and *manual test case generation*. The first approach yields a great number of test cases in a short time and at a relatively low cost. In most cases, as originally proposed by Purdom [17], test programs are derived from a grammar of the source language by systematically exercising all its productions. An overview of this and related approaches is given in [6]. However, the quality of the test cases is questionable because the test case generation process is not guided by the requirements (i.e. the specification).

A different (and more reliable) method is to generate test cases manually with respect to given language standards, like the Ada Conformity Assessment Test Suite (ACATS)[5], or commercial testsuites for ANSI/ISO C language conformance[6]. However, there is no published standard for graphical source languages such as Simulink or Stateflow. Moreover, the manual creation and maintenance of test cases is cost-intensive, time-consuming and also requires knowledge about tool internals.

A technique for testing a code generator systematically on the basis of graph rewriting rules was proposed in [19]. The graph-rewriting rules *themselves* are

[5] See `www.adaic.com`

[6] ANSI/ISO FIPS-160 C Validation Suite (ACVS) by Perennial, `www.peren.com`

used as a blueprint for systematic test case (i.e. model) generation. Additionally, the mentioned paper shows how such models can be used in practice. A two-level hierarchy of testing is proposed: First, suitable input models to be used as test cases for the code generator are determined; then the behaviour of the code generated from such models over specific (suitably chosen) input data is compared with that of the (executable) specification, in order to ensure correctness. A difference with the work in the present paper is that in [19] a test case is selected on the basis of a single rule, while here we will consider the interaction of several rules to derive test cases which can trigger these complex behaviours. Still the methodology proposed in [19] to use the test cases once they are available, can be applied also to the test cases produced via the technique in our paper.

Graph transformation systems have also been used in other ways in connection with code generator specification or verification. For instance, in [11, 16] graph and tree replacement patterns are used for verifying a code generator formally and in [2] graph rewriting rules are used for generating an optimiser. A complete code generator, capable of translating Simulink or Stateflow models into C code, has been specified in [14] with the Graph Rewriting and Transformation language GReAT [12].

3 Graph Transformation Systems

We use hypergraphs, which allow us to conveniently represent functions with n arguments by $(n+1)$-ary hyperedges (one connection for the result, the rest for the parameters). Moreover we use graph rewriting rules as in the double-pushout approach [8, 10] with added negative application conditions [9]. A rule, apart from specifying a left-hand side graph that is removed and a right-hand side graph that replaces it, specifies also a context graph that is preserved, and forbidden edges that must not occur attached to the left-hand side.

Hereafter Λ is a fixed set of edge *labels* and each label $l \in \Lambda$ is associated with an *arity* $ar(l) \in \mathbb{N}$. Given a set A, we denote by A^* the set of finite sequences of elements of A and for $s \in A^*$, $|s|$ denotes its length.

Definition 1 (Hypergraph). *A* (Λ-)hypergraph G *is a tuple* (V_G, E_G, c_G, l_G)*, where* V_G *is a set of nodes,* E_G *is a set of edges,* $c_G\colon E_G \to V_G^*$ *is a connection function and* $l_G\colon E_G \to \Lambda$ *is the labelling function for edges satisfying* $ar(l_G(e)) = |c_G(e)|$ *for every* $e \in E_G$*. Nodes are not labelled.*

Hypergraph morphisms $\varphi\colon G \to G'$ and isomorphisms are defined as usual.

Definition 2 (Graph rewriting rules with negative conditions). *A* graph rewriting rule r *is a tuple* $(L \xleftarrow{\varphi_L} I \xrightarrow{\varphi_R} R, N)$ *where* $\varphi_L\colon I \to L$ *and* $\varphi_R\colon I \to R$ *are injective graph morphisms. We call* L *the* left-hand side, R *the* right-hand side *and* I *the* context*. We assume that (i)* φ_L *is bijective on nodes, (ii)* L *does not contain isolated nodes, (iii) any node isolated in* R *is in the image of* φ_R*.*

Furthermore N *is a set of injective morphisms* $\eta\colon L \to L_\eta$*, called* negative application conditions*, where (iv)* $E_{L_\eta} - \eta(E_L)$ *contains a single edge referred to as* e_η *and (v)* L_η *does not contain isolated nodes.*

A rule $r = (L \overset{\varphi_L}{\leftarrow} I \overset{\varphi_R}{\rightarrow} R, N)$ consists of two components. The first component $L \overset{\varphi_L}{\leftarrow} I \overset{\varphi_R}{\rightarrow} R$ is a graph production, specifying that an occurrence of the left-hand side L can be rewritten into the right-hand side graph R, preserving the context I. Condition (i) stating that φ_L is bijective on nodes ensures that no nodes are deleted. Nodes may become disconnected, having no further influence on rewriting, and one can imagine that they are garbage-collected. Actually, Conditions (ii) and (iii) essentially state that we are interested only in rewriting up to isolated nodes. By (iii) no node is isolated when created and by (ii) nodes that become isolated have no influence on further reductions.

The second component N is the set of negative application conditions. Intuitively, each L_η extends the left-hand side L with an edge e_η which must not be connected to the match of L to allow a rule to be applied. The negative application conditions here are weaker than in [9]. This will allow us to represent negative application conditions by inhibitor arcs in the unfolding (see Section 5).

Definition 3 (Match). *Let $r = (L \overset{\varphi_L}{\leftarrow} I \overset{\varphi_R}{\rightarrow} R, N)$ be a graph rewriting rule and let G be a graph. Given an injective morphism $\varphi\colon L \to G$, a falsifying extension for φ is an injective morphism $\varphi'\colon L_\eta \to G$ such that $\varphi' \circ \eta = \varphi$ for some $\eta \in N$. In this case $\varphi'(e_\eta)$ is called a* falsifying edge *for φ. The morphism φ is called a* match *of r whenever it does not admit any falsifying extension.*

Given a graph G and a match in it, G can be rewritten to H (in symbols: $G \Rightarrow H$), by applying rule r as specified in the double-pushout approach [8].

Definition 4 (Graph grammar). *A* graph grammar $\mathcal{G} = (\mathcal{R}, G_0)$ *consists of a set of rewriting rules $\mathcal{R}$ and a* start graph G_0 *without isolated nodes. We say that a graph G is generated by $\mathcal{G}$ whenever $G_0 \Rightarrow^* G$.*

4 Specifying Code Generation by Graph Transformation

In our setting, code generation starts from an internal graph representation of a Simulink or Stateflow model, free of layout information. Especially the first steps of code generation, involving optimisations that change the graph structure of a model (e.g. for dead code elimination), can be naturally described by graph rewriting rules. In the sequel, the set of optimising rules is called *optimising grammar*, even if we do not fix a start graph. Since our aim is to test the code generator itself, independently of a specific Simulink model, we need some means to describe the set of all possible models that can be given as input to the code generator. In our proposal this is seen as a graph language generated by another grammar, called *generating grammar*.

Example: We illustrate the above concepts with an example describing the first steps of code generation, starting from acyclic graphs which represent arithmetic expressions. We will give only excerpts of the two graph rewriting systems: the generating grammar, describing acyclic graphs which represent arithmetic expressions, and the optimising grammar, describing constant folding, i.e., simplification and partial evaluation of arithmetic expressions.

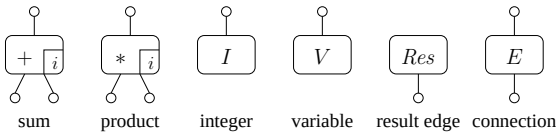

Fig. 1. Edge types for the graph rewriting system (constant folding).

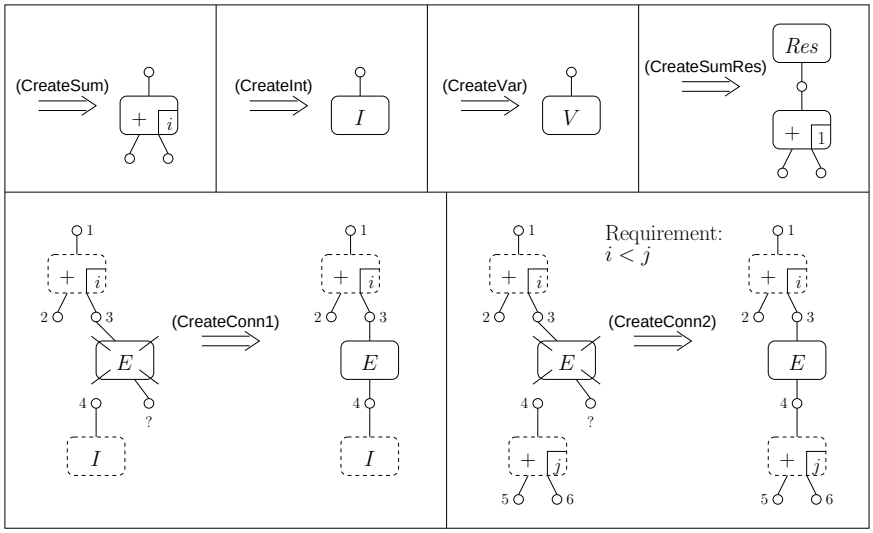

Fig. 2. Rules of the generating grammar (constant folding).

We assume that a maximal depth a is fixed for arithmetic expressions. Then we will use the edge types depicted in Fig. 1 for $1 \leq i \leq a$. Integers and variables are generically represented by I and V edges. Since in our setting we are mainly concerned with structural optimisation steps, we do not consider attributes here: As soon as a test case is generated, it can be equipped with suitable values for all the constants involved. For instance, Fig. 3 shows an acyclic graph representing the arithmetic expression $i_1 + (i_1 * i_2)$ for some arbitrary integers i_1, i_2.

The *generating grammar* $\mathcal{G}_{\mathbf{g}}$, which is depicted in Fig. 2, generates operator, result, integer and variable edges and connects them via E-edges (connecting edges), provided no edge of this kind is present yet. The rules are specified in the form "left-hand side $\Rightarrow$ right-hand side". Edges of the context are drawn with dashed lines and nodes of the context are marked with numbers. Negative application conditions are depicted as crossed-out edges. Note that (CreateConn2) is a rule schema: an E-edge between operator edges is only allowed if the first operator has a smaller (arithmetic) depth than the second one, i.e., if $i < j$, thus ensuring acyclicity. Some of the rules are missing, for example the rule generating a product edge (analogous to the rule (CreateSum)) and several more rules connecting operator edges. The start graph is the empty graph.

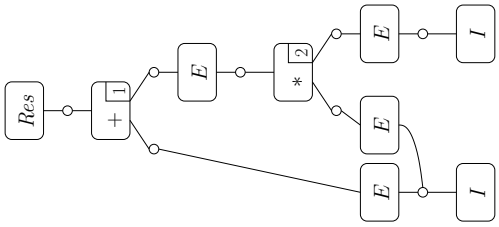

Fig. 3. A graph representing the arithmetic expression $i_1 + (i_1 * i_2)$.

The *optimising grammar* $\mathcal{G}_\mathsf{o}$ is (partially) presented in Fig. 4: We give rules for reducing the sharing of constants (ConstantSplitting), for removing useless or isolated parts of the graph (KillUselessFunction), (KillLonelyEdge), and for simplifying the graph by evaluating the sum of two integers (ConstantFoldingSum). More specifically, the optimisation corresponding to the last rule computes the integer of the right-hand side as the sum of the two integers of the left-hand side. A requirement for its application is the absence of constant sharing.

5 Unfolding Graph Transformation Systems

The unfolding approach, originally devised for Petri nets [15], is based on the idea of associating to a system a single branching structure, representing all its possible runs, with all the possible events and their mutual dependencies. For graph rewriting systems, the unfolding is constructed starting from the start graph, considering at each step the rewriting rules which could be applied and then recording in the unfolding possible rule applications and the graph items which they generate (see [18, 4]). Here we sketch how the unfolding construction can be extended to graph grammars with negative application conditions. Space limitations keep us from giving a formal presentation of the theory.

The unfolding of a graph grammar will be represented as a Petri net-like structure. We next introduce the class of Petri nets which plays a basic role in the presentation. Given a set S, we denote by $S^\oplus$ the set of multisets over S, i.e., $S^\oplus = \{m \mid m : S \to \mathbb{N}\}$. A multiset m can be thought of as a subset of S where each $s \in S$ occurs with a multiplicity $m(s)$. When $m(s) \in \{0, 1\}$ for all $s \in S$ the multiset m will often be confused with the set $\{s \in S \mid m(s) = 1\}$.

Definition 5 (Petri net with read and inhibitor arcs). *Let L be a set of transition labels. A* Petri net with read and inhibitor arcs *is a tuple $N = (S_N, T_N, {}^\bullet(), ()^\bullet, \underline{()}, {}^\circledcirc(), p_N)$ where S_N is a set of* places*, T_N is a set of* transitions *and $p_N : T_N \to L$ is a labelling function. For any transition $t \in T_N$, ${}^\bullet t$, $t^\bullet$, $\underline{t}$, ${}^\circledcirc t \in {S_N}^\oplus$ denote pre-set, post-set, context and set of inhibitor places of t.*

When $s \in \underline{t}$ we say that t is connected to s via a *read arc*. In this case t may only fire if place s contains a token. This token will not be affected by the firing. On the other hand, when $s \in {}^\circledcirc t$ we say that t is connected to s via an *inhibitor*

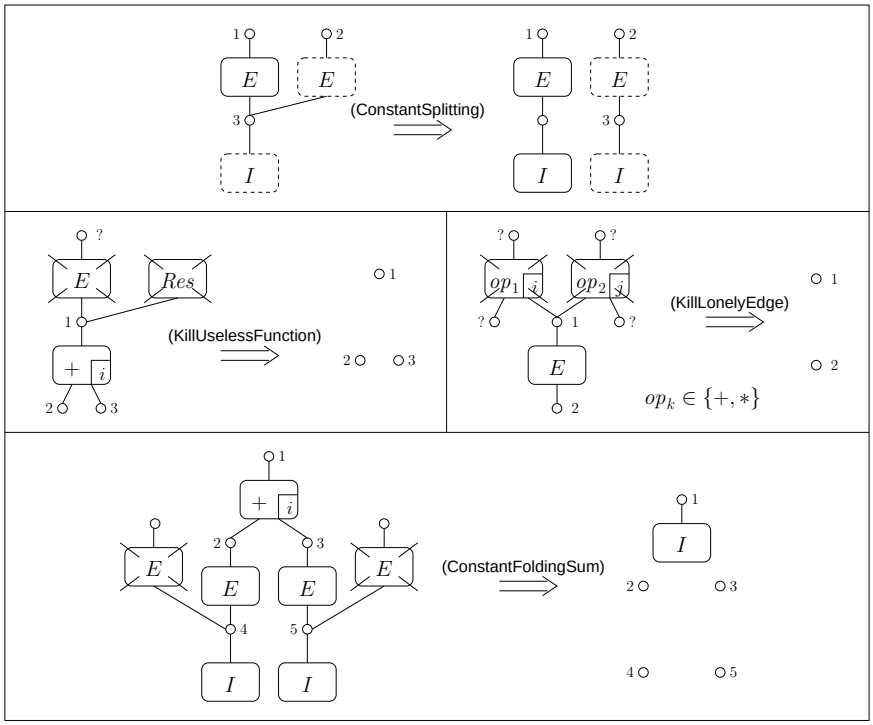

Fig. 4. Rules of the optimising grammar (constant folding).

arc and t is allowed to fire only if s does *not* contain a token. Read arcs [13] will be used to represent, at the level of Petri nets, the effects arising from the possibility of preserving graph edges in a rewriting step. Inhibitor arcs [1] will be used to model the effects of the negative application conditions that we have at the level of graph grammar rules.

The mutual dependencies between transitions play a crucial role in the definition of the unfolding. Given a net N, the *causality* relation $<_N$ is the least transitive relation such that $t_1 <_N t_2$ if $t_1{}^\bullet \cap ({}^\bullet t_2 \cup \underline{t_2}) \neq \emptyset$, i.e., if t_1 produces a token consumed or read by t_2.

In ordinary Petri nets, two transitions t_1 and t_2 competing for a resource, i.e., which have a common place in the pre-set, are said to be in conflict. The presence of read arcs leads to an *asymmetric* form of conflict: if a transition t_2 "consumes" a token which is "read" by t_1 then the execution of t_2 prevents t_1 to be executed, while the sequence "t_1 followed by t_2" is legal. The *asymmetric conflict* $\nearrow_N$ can be formally defined by $t_1 \nearrow_N t_2$ if $\underline{t_1} \cap {}^\bullet t_2 \neq \emptyset$ or ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$. The last clause includes the ordinary symmetric conflict as asymmetric conflict in both directions, i.e., $t_1 \nearrow_N t_2$ and $t_2 \nearrow_N t_1$. More generally, transitions occurring in a cycle of asymmetric conflicts $t_1 \nearrow_N t_2 \nearrow_N \ldots \nearrow_N t_n$ cannot appear in the same computation since each of them should precede all the others.

The unfolding of a graph grammar with negative application conditions is defined as a Petri graph [5], i.e., a graph with a Petri net "over it", using the edges of the graph as places.

Definition 6 (Petri graph). *Let $\mathcal{G} = (\mathcal{R}, G_0)$ be a graph grammar. A Petri graph (over $\mathcal{G}$) is a tuple $P = (G, N)$ where G is a hypergraph, N is a Petri net with read and inhibitor arcs whose places are the edges of G, i.e., $S_N = E_G$, and the labelling $p_N : T_N \to \mathcal{R}$ of the net maps the transitions to the graph rewriting rules of $\mathcal{G}$. A Petri graph with initial marking is a tuple (P, m_0) where $m_0 \in {E_G}^{\oplus}$.*

Each transition in the Petri net will be interpreted as an occurrence of a graph production at a given match. Note that Definition 6 does not ask that the pre-set, post-set, context or inhibitor places of a transition t have any relation with the corresponding graph rewriting rule $p_N(t)$, but the unfolding construction presented later will ensure a close relation.

As in Petri net theory, a marking $m \in {E_G}^{\oplus}$ is called *safe* if any place (edge) contains at most one token. A safe marking m of a Petri graph $P = (G, N)$ can be seen as a graph, i.e., the least subgraph of G including exactly the edges which contain a token in m. Such a graph, denoted by $graph(m)$, is called the *graph generated* by m.

We next describe how a suitable unfolding can be produced from the generating/optimising grammars associated to a code generator. We introduce the criteria and conditions that must be met step by step. At first, the graph grammars are unfolded disregarding the negative application conditions.

Petri graph corresponding to a rewriting rule: Every graph rewriting rule can be represented as a Petri graph without considering negative application conditions: Take both the left-hand side L and the right-hand side R, merge edges and nodes that belong to the context and add a transition, recording which edges are deleted, preserved and created. For instance the Petri graph P corresponding to rule (CreateConn1) in Fig. 2 is depicted in Fig. 5 (see the Petri graph in the middle). Observe that the transition preserves the edges labelled + and I (read arcs are indicated by undirected dotted lines) and produces an edge labelled E. In this case no edges are deleted. In order to distinguish connections of the graph and connections between transitions and places, we draw the latter as dashed lines.

Unfolding step: The initial Petri graph is obviously the start graph G_0 of the generating grammar, with no transitions. At every step, we first search for a match of a left-hand side L, belonging to a graph production r. This match must be potentially coverable (concurrent), i.e., it must not contain items which are causally related and the set of causes of the items in the match must be conflict-free (negative application conditions will be taken into account later). We now take the Petri graph P associated to r and merge the edges and nodes of L in P with the corresponding items of the occurrence of L in the partial unfolding. Fig. 5 exemplifies this situation for an incomplete unfolding $\mathcal{U}$ and the Petri graph P representing rule (CreateConn1) of Fig. 2. The left-hand side L which indicates how the merging is to be performed, is marked in grey.

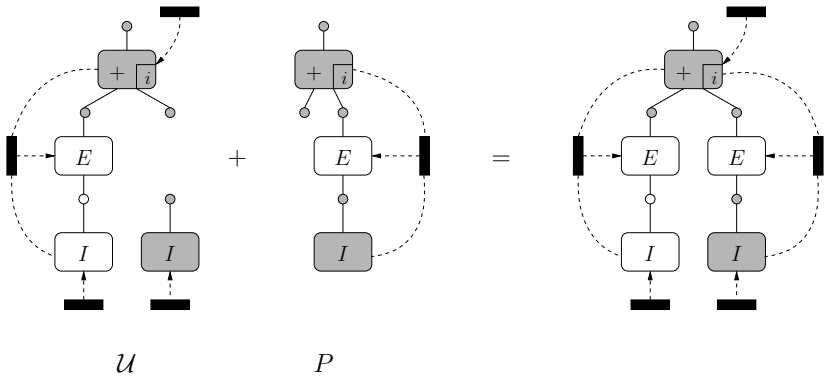

Fig. 5. Performing an unfolding step.

The unfolding of a given graph grammar is usually infinite and thus its construction could continue for an arbitrarily long time. To avoid this we employ two mechanisms, called depth restriction and width restriction.

Depth restriction: The idea consists of truncating the unfolding after a certain depth has been reached. To make this more formal, define the *depth of a transition* t to be the length of the longest sequence $t_0 <_N t_1 <_N \ldots <_N t_n <_N t$, where $<_N$ is the causality relation. The depth of an edge is the depth of the unique transition having such an edge in its post-set. If the edge is in the start graph, then its depth is set to 0. For our purposes it is not necessary to define the depth of a node. Then we fix a parameter k, called *depth restriction*, asking that no items of depth greater than k are ever created by the unfolding construction.

Width restriction: Depth restriction is not sufficient to keep the unfolding finite, since matches of a left-hand side could be unfolded more than once. To stop after a finite number of steps we impose the following conditions:

(1) A rule r which deletes at least one edge, i.e., for which the left-hand side is strictly larger than the context, is applied only once to every match. Note that we would not gain anything from unfolding such a match twice: since at least one token is consumed by firing the corresponding transition and the unfolding is acyclic, the pre-set could never be covered again and thus it would not be possible to fire another copy of the same transition.

(2) A rule r which does not delete any edge, i.e., for which the left-hand side is equal to the context, is unfolded w times for every match, where w is a fixed parameter called *width restriction*. The different copies of this transition can potentially all be fired since no edge is ever consumed. Actually, since each copy of the transition has an empty pre-set it could possibly be fired more than once, leading to more than one token in a place, a situation which must be avoided in the unfolding where each transition is intended to represent a single occurrence of firing and each place a single occurrence of a token. This problem is solved by

transition	rule	depth
t_1	CreateSum	1
t_2	CreateSumRes	1
t_3, t_4	CreateInt	1
t_5, t_6	CreateConn	2

(a) Grammar $\mathcal{G}_\mathsf{g}$

transition	rule	depth
t_8	ConstantSplitting	3
t_9	KillUselessFunction	2
t_{10}	KillLonelyEdge	3
t_{11}	ConstantFoldingSum	3

(b) Grammar $\mathcal{G}_\mathsf{o}$

Table 1. Correspondence between transitions and rules.

introducing a dummy place—initially marked—as the pre-set of such transitions, ensuring that every transition is fired only once.

Generating grammar before optimising grammar: We still have to avoid mixing the two grammars. So far it is still possible to create an unfolding that includes sequences of rewriting steps where rules of the generating grammar are applied to the start graph, followed by the application of rules of the optimising grammar and then again by rules of the generating grammar. Derivations of this kind do not model any interesting situation: in practice, first the model is created, and only then are optimising steps allowed. Hence we impose that whenever there are transitions t_1 and t_2 such that $t_1 <_N t_2$ and $p_N(t_2)$ is a rule of the generating grammar, then also $p_N(t_1)$ must be a rule of the generating grammar. In such a situation we say that $<_N$ is *compatible with the grammar ordering*.

Add inhibitor arcs: In the next step, the final unfolding is obtained by taking every transition t in the Petri graph, labelled by a rule r, considering the corresponding match and adding, for any falsifying edge, an inhibitor arc. Inhibitor arcs are represented by dotted lines with a small circle at one end.

Initial marking: The initial marking contains exactly the edges of the start graph and, in addition, all dummy places that were created during the unfolding.

The structure produced by the above procedure is referred to as *unfolding up to depth k and width w* and denoted by $\mathcal{U}_k^w$.

Example (continued): Fig. 6 shows a part of the unfolding for the grammars of the running example. We assume that the depth restriction k is at least 3, the width restriction is at least 2 and the arithmetic depth a is also at least 2. Table 1 shows the labelling of transitions over rewriting rules and the causal depth of each transition. The depth of every dummy place is 0, while the depth of any other place (edge) is the depth of the transition which has this edge in its post-set. Note that two inhibitor arcs at transitions t_{10} and t_{11} in Fig. 6 are inserted because of the presence of falsifying edges.

The unfolding faithfully represents system behaviour in the following sense.

Proposition 1. *Let G_0 be the start graph of the generating grammar and let $G_0 \Rightarrow^* G$ be a derivation of G such that: (i) the derivation consists of at most k*

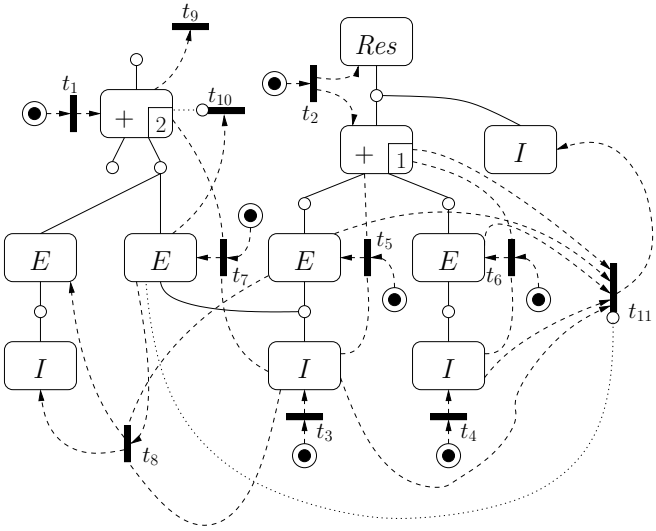

Fig. 6. A part of the unfolding for the example grammars (constant folding).

(possibly concurrent) steps, (ii) no rewriting rule is applied more than w times to the same match, (iii) rules of the optimising grammar are applied only after those of the generating grammar. Then there is a reachable marking m in the unfolding truncated at depth k and width w, such that G is isomorphic to graph(m) up to isolated nodes. Furthermore, for every reachable marking m there is a graph G such that $G_0 \Rightarrow^* G$ and G is isomorphic to graph(m), up to isolated nodes.

6 Generating Test Cases

The application order of optimisation techniques is not fixed a priori, but depends on specific situations within the input graph. Hence, with respect to testing, the situation is quite different from that of imperative programs for which there exists a widely accepted notion of coverage. In order to achieve in our case an adequate coverage for possible optimisation applications (of a single optimisation rule or a combination of different optimisation techniques) we propose to derive (graphical) input models which trigger the application of a single optimisation step or trigger the "combination" of several rules. In the last case the occurrences of the selected rules should be causally dependent on each other or in asymmetric conflict, such that error-prone interactions of rules can be tested.

Test cases triggering such a behaviour can be derived from the unfolding (up to depth k and up to width w) which provides a very compact description of all graphs which can be reached and of all rules which can be applied in a certain number of steps (see Proposition 1).

In the following, we denote by $\mathcal{R}_\mathsf{g}$ the set of rules of the generating grammar, by G_s its start graph and by $\mathcal{R}_\mathsf{o}$ the set of optimising rules.

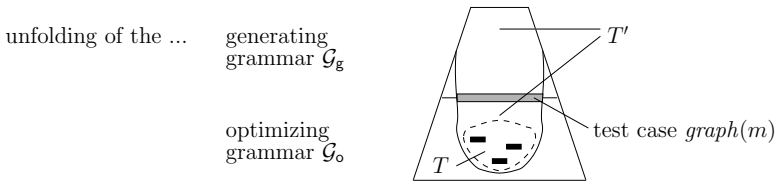

Fig. 7. Schematical representation of test case generation.

Definition 7 (Test case). *Given a set of optimising rules $R \subseteq \mathcal{R}_{\mathtt{o}}$, a* test case *for R is an input model G such that a computation of the optimiser over G can use all the rules in R.*

Take a set R of interesting rules the interaction of which should be tested. The set R can be determined by the tester or by general principles, for instance one could take all sets R up to a certain size. Then proceed as follows:

(1) Take a set T of transitions in $\mathcal{U}_k^w$ (the unfolding up to depth k and up to width w) labelled by rules in R such that (a) for all $r \in R$, there exists a transition $t_r \in T$ such that t_r is labelled by r; (b) for all transitions $t_r \in T$ there exists a transition $t'_r \in T$ which is related to t_r by asymmetric conflict or causal dependency.
(2) Look for a set T' of transitions in $\mathcal{U}_k^w$ such that $T \subseteq T'$, and exactly the transitions of T' can be fired in a derivation of the grammar. Note that not every set T can be extended to such a T', since transitions might be in conflict or block each other by inhibitor arcs.
(3) Take the subset of transitions in T' labelled by rules in $\mathcal{R}_{\mathtt{g}}$ and fire such rules, obtaining a marking m. Then $graph(m)$ is a test case for R.

See Fig. 7 for a schematical representation of the above procedure. Whenever the specification is non-deterministic, we cannot guarantee that the execution over the test case really involves the transformation rules in R, but this is a problem inherent to the testing of non-deterministic systems.

The set T' can be concretely defined by resorting to the notions of configuration and history in the theory of inhibitor Petri nets [7, 3]. Roughly, a *configuration* of $\mathcal{U}_k^w$ is a pair $\langle C, <_C \rangle$ where C is a set of transitions closed under causality and $<_C$ is a partial order including causality $<_{\mathcal{U}}$, asymmetric conflict $\nearrow_{\mathcal{U}}$ and a relation $<_p$ which considers the effects of inhibitor arcs: For any place s connected to a transition $t \in C$ by means of an inhibitor arc ($s \in {}^{\circledcirc}t$) it chooses if t is executed before the place is filled or after the place is emptied. A configuration $\langle C, <_C \rangle$ can be seen as a concurrent computation, $<_C$ being a computational ordering on transitions, in the sense that the transitions in C can be fired in any total order compatible with $<_C$. A configuration $\langle C, <_C \rangle$ is called *proper* if the partial order $<_C$ is compatible with the grammar ordering, i.e., if $t_1 <_C t_2$ and t_2 belongs to the generating grammar then also t_1 belongs to the generating grammar.

The *history* of a transition t in a configuration $\langle C, <_C \rangle$, denoted by $C[t]$ is the set of transitions which must precede t in any computation represented by C. Formally, $C[t] = \{t' \in C \mid t' \leq_C t\}$. Note that a transition t can have several possible histories in different configurations. This is caused by the presence of read arcs and, even more severely, by inhibitor arcs. With asymmetric conflict only, there is a least history, the set of (proper) causes $\lfloor t \rfloor = \{t' \mid t' \leq_N t\}$, and the history of an event in a given configuration is completely determined by the configuration itself. With inhibitor arcs, in general, there might be several histories of a transition in a given configuration, and even several minimal ones.

Hence, coming back to the problem in step (2) above, the set T' we are looking for can be defined as a proper configuration including T. The choice among the possible configurations T' including T could be influenced by the actual needs of the tester. In many cases the obvious choice will be to privilege configurations with minimal cardinality, since these contain only the events which are strictly necessary to make the rules in T applicable.

Example (continued): We continue with our running example. Assume that we want a test case including the application of rule (ConstantFoldingSum), i.e., $R = \{(\mathsf{ConstantFoldingSum})\}$ (the procedure works in the same way for more than one rule). Transition t_{11} is an instance of this rule and it is contained in several different configurations, for example $H_1 = \{t_2, t_3, t_4, t_5, t_6, t_{11}\}$ which creates only a sum with two integers and corresponding connections, $H_2 = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_{11}\}$ which creates another E-edge and removes it by (ConstantSplitting) and $H_3 = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_9, t_{10}, t_{11}\}$ which creates another E-edge and another sum and removes them by (KillUselessFunction) and (KillLonelyEdge). All these configurations are histories of t_{11}. Depending on the choice of the history, one obtains the two different test cases (the first for H_1 and the other for H_2 and H_3) in Fig. 8.

To understand how such a test case is derived, consider the history H_3. Causality ($<_\mathcal{U}$), asymmetric conflict ($\nearrow_\mathcal{U}$) and the relation $<_p$ in H_3 are depicted in Fig. 9. Note, for instance, that t_7 is forced to fire before t_{11}, since t_7 corresponds to a rule of grammar $\mathcal{G}_\mathbf{g}$, while t_{11} is labelled with a rule of $\mathcal{G}_\mathsf{o}$. After firing t_7, transition t_{11} is blocked by an inhibitor arc, and t_{11} can only be enabled by firing t_9 and t_{10}. Thus $t_9 <_p t_{10} <_p t_{11}$ is the only possible choice. Furthermore t_7 and t_9 are in asymmetric conflict ($t_7 \nearrow_N t_9$), since t_9 removes an element of the context of t_7. By taking the subset of rules in H_3 which belong to the generating grammar $\mathcal{G}_\mathbf{g}$, namely $t_1, t_2, t_3, t_4, t_5, t_6, t_7$, and firing them, we obtain the test case on the right-hand side of Fig. 8.

7 Conclusion

We have presented a technique for deriving test cases for code generators with a graphical source language. The technique is based on the formalisation of code generators by means of graph transformation rules and on the use of (variants of the) unfolding semantics as a compact description of their behaviour.

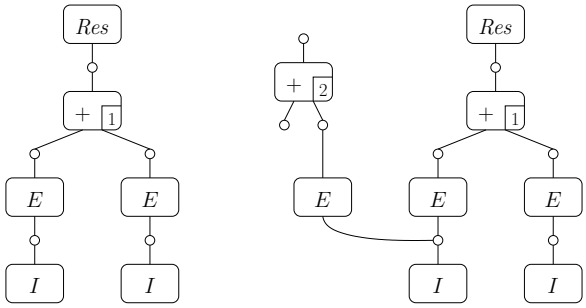

Fig. 8. Generated test cases.

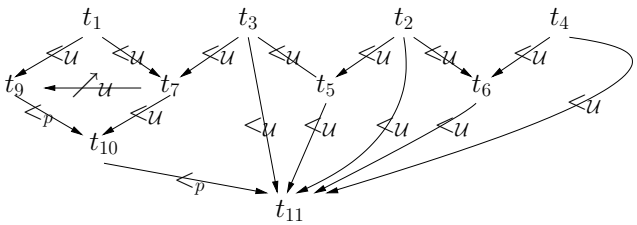

Fig. 9. Causality, asymmetric conflict and relation $<_p$.

The novelty of our approach consists in the fact that we consider graphical models and that we generate a compact description of system behaviour from which we can systematically derive test cases triggering specific behaviours. By using an unfolding technique we can avoid considering all interleavings of concurrent events, thereby preventing combinatorial explosion to a large extent. We believe that this technique can also be very useful for testing programs of visual programming languages.

This paper does not address efficiency issues. Note that the causality relation and asymmetric conflict of an occurrence net can be computed statically without firing the net. Hence, it is important to note that without inhibitor arcs, configurations and histories and hence test cases can be determined in a very efficient way. In the presence of inhibitor arcs, it is necessary to construct suitable relations $<_p$, leading to configurations. Obtaining such relations $<_p$ is quite involved and requires efficient heuristics, which we have already started to develop in view of an upcoming implementation of the test case generation procedure.

Acknowledgements: We would like to thank the anonymous referees for their helpful comments. We are also grateful to Andrea Corradini for comments on an earlier version of this paper.

References

1. T. Agerwala and M. Flynn. Comments on capabilities, limitations and "correctness" of Petri nets. *Computer Architecture News*, 4(2):81–86, 1973.
2. U. Assmann. Graph rewrite systems for program optimization. *TOPLAS*, 22(4):583–637, 2000.
3. P. Baldan, N. Busi, A. Corradini, and G.M. Pinna. Functorial concurrent semantics for Petri nets with read and inhibitor arcs. In *CONCUR'00 Conference Proceedings*, volume 1877 of *LNCS*, pages 442–457. Springer Verlag, 2000.
4. P. Baldan, A. Corradini, and U. Montanari. Unfolding and event structure semantics for graph grammars. In *Proceedings of FoSSaCS '99*, volume 1578 of *LNCS*, pages 73–89. Springer Verlag, 1999.
5. P. Baldan and B. König. Approximating the behaviour of graph transformation systems. In *Proc. of ICGT '02 (International Conference on Graph Transformation)*, pages 14–29. Springer-Verlag, 2002. LNCS 2505.
6. A.S. Boujarwah and K. Saleh. Compiler test case generation methods: a survey and assessment. *Information and Software Technology*, 39(9):617–625, 1997.
7. N. Busi and G.M. Pinna. Non sequential semantics for contextual P/T nets. In *Application and Theory of Petri Nets*, volume 1091 of *LNCS*, pages 113–132. Springer Verlag, 1996.
8. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation—part I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations*, chapter 3. World Scientific, 1997.
9. H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation—part II: Single pushout approach and comparison with double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations*, chapter 4. World Scientific, 1997.
10. H. Ehrig, M. Pfender, and H. Schneider. Graph grammars: An algebraic approach. In *Proc. 14th IEEE Symp. on Switching and Automata Theory*, pages 167–180, 1973.
11. S. Glesner, R. Geiß, and B. Boesler. Verified code generation for embedded systems. In *In Proceedings of the COCV-Workshop (Compiler Optimization meets Compiler Verification)*, volume 65.2 of *ENTCS*, 2002.
12. G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle. On the use of graph transformation in the formal specification of model interpreters. *Journal of Universal Computer Science*, 9(11):1296–1321, 2003.
13. U. Montanari and F. Rossi. Contextual nets. *Acta Informatica*, 32(6), 1995.
14. S. Neema and G. Karsai. Embedded control systems language for distributed processing (ECSL-DP). Technical Report ISIS-04-505, Vanderbilt University, 2004.
15. M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains, Part 1. *Theoretical Computer Science*, 13:85–108, 1981.
16. A. Nymeyer and J.-P. Katoen. Code generation based on formal BURS theory and heuristic search. *Acta Informatica*, 34:597–635, 1997.
17. P. Purdom. A sentence generation for testing parsers. *BIT*, pages 366–375, 1972.
18. L. Ribeiro. *Parallel Composition and Unfolding Semantics of Graph Grammars.* PhD thesis, Technische Universität Berlin, 1996.
19. I. Stürmer and M. Conrad. Test suite design for code generation tools. In *Proc. 18th IEEE Automated Software Engineering (ASE) Conference*, pages 286–290, 2003.