A Graph Grammar to Transform a Dataflow Graph into a Multithread Graph and its Application in Task Scheduling

Simone André da Costa Cavalheiro [1]
Luciana Foss [1]
Cícero Augusto de S. Camargo [1]
Gerson Geraldo H. Cavalheiro [1]

Abstract: The scheduling of tasks in a parallel program is a NP-complete problem, where scheduling tasks over multiple processing units requires an effective strategy to maximize the exploitation of the parallel hardware. Several studies focus on the scheduling of parallel programs described as DAGs (Directed Acyclic Graphs). However, many modern multithread environments can get high performance levels using a lighter representation to describe the program, the DCG (Directed Cyclic Graph). This paper shows the structure and semantics of a DCG, and proposes patterns to map structures found in DAGs into segments of a DCG. A graph grammar has been developed to perform the proposed transformation and case studies using DAGs found in the literature validate the transformation process. Besides the automatic translation and precise definition of the mapping, the use of a formal language also allowed the verification of the existence and uniqueness of the outcoming model.

1 Introduction

Scheduling algorithms are used in parallel programming to allocate program tasks over the available processors on a parallel architecture [1]. The main goal of scheduling is assigning a starting time and a processor to each task generated by a parallel program. The scheduler must also guarantee that the program will be completed in a finite time. Scheduling techniques often include optimization goals, such as minimizing execution time or memory usage of a program. Scheduling algorithms can also be applied at application level [2] so they can consider characteristics of the program being executed for decision making.

In parallel programming, a well-known model used to express concurrency is the dataflow model [3]. Programs that are modeled according to the dataflow model are suitable to be described in a Directed Acyclic task Graph (DAG), where vertices represent tasks and edges represent data dependencies between two tasks. In this model each task describes a sequence of instructions to be computed and each data dependence describes a communication of data between two tasks. Thus, the set of edges in a dataflow program represent the precedence constraints to execute the tasks. At execution time, two tasks can

[1]Programa de Pós-Graduação em Computação (PPGC), Universidade Federal de Pelotas, Campus Porto
{simone.costa,lfoss,cadscamargo,gerson.cavalheiro}@inf.ufpel.edu.br

be executed in arbitrary order or at the same time on different processors only if there are no precedence restrictions among them. Otherwise, the data communication requirements impose a specific execution order between those two tasks.

Graham [4] presented the scheduling bounds for static DAGs. In that case the whole graph is known a priori and the complete scheduling can be set before the program starts. The scheduling of DAGs is based on the list scheduling technique where the basic idea is to make a list of tasks by assigning them some priority [5]. Kwok and Ahamad [5] propose a taxonomy for DAG scheduling algorithms as well as they present the basic techniques to schedule DAGs. In general, scheduling a DAG is an NP-complete problem [6]. We can found in [7] a solution that takes a polynomial-time for three case studies and in [8] a linear-time algorithm to schedule DAGs. The Dominant Sequence Clustering (DSC) [9] also presents optimal performance considering special classes of applications.

Most recently, we have found in the literature many execution environments, such as Intel® Cilk Plus [10], OpenMP [11], and Intel®Threading Building Blocks [12], that apply dynamic scheduling strategies based on list scheduling to support the execution of multithreaded programs. These environments provide programming abstractions to describe a concurrent program in terms of a DAG. They also include scheduling heuristics to assign different priorities of execution for the tasks considering their relation to the Critical Path of the DAG. Practical performance results indicate that we must minimize the overheads implied by the environment's operations, mainly for operations that manipulate tasks in the Critical Path.

Many academic multithread programming tools, such as Anahy [13] and KAAPI [14], also use strategies based on static DAG scheduling to schedule dynamic multithreaded programs. In this scenario programs respect the multithread model [15], which represents programs in a different way from the dataflow model: a multithreaded program can be represented in a Directed Cyclic Graph (DCG), where vertices represent threads and edges represent *create* and *join* operations. However, threads just encapsulate sequences of tasks in a proper way, so that a DAG can be obtained apart from a DCG, even on dynamic scenarios. The opposite way, obtaining a DCG from a DAG, is also possible, that is, we can get a multithreaded program from a given task graph.

This work proposes a formal model to map a DAG into an equivalent DCG, representing the same program written in a multithread programming interface using only *create* and *join* primitives. The use of a formal language to define the translation avoids possible ambiguities from a natural language description. Since the source and target of the mapping are graphs, it is natural to consider that the translation from DAGs to DCGs are based on rules that transform graphs. Graph grammars are a formal language that follow such approach [16], and offer various results concerning different types of analysis (like termination and confluence) that are suitable for model transformations. Besides,

adopting graph grammars, it is possible to automate the translation process and to have support for confluence analysis by using the Attributed Graph Grammar system (AGG tool) [17, 18]. It would be possible to model the mapping with some programming language or pseudocode, however in such case we would have to map graphs to a concrete data structure. The adoption of the graph grammar language avoids this translation and simplifies the specification. Moreover, the use of such language also allows, in the future, to apply available techniques and tools for formal verification. This mapping allows measuring the efficiency of scheduling techniques that consider different graph representations. The present paper extends the results presented in [19]. With respect to the original version, we included two relevant contributions:

1. The exemplification of the proposed transformation to several case studies found in literature as well as its comparison with previous scheduling results: several tests were run with DAGs as input parameters, all resulting in successful mappings to equivalent DCGs;
2. The proofs of termination and confluence of the translation: the graph grammar specification also allowed the formal analysis of the mapping. Termination ensures that the transformation process finishes, while confluence ensures that the transformation results in a unique target model.

The remaining of this paper is organized as follows. Section 2 reviews some concepts on attributed graph grammars, used in the translation process of DAGs into DCGs, while Section 3 details the semantics of DAGs and DCGs, and the translation process itself. Section 4 details the transformation analysis and Section 5 presents some case studies. Concluding remarks are defined in Section 6.

2 Graph Grammar

In this section we review the main concepts about typed attributed graph grammars with application condition and negative application conditions, based on the double pushout approach (DPO-approach) [16]. Basically, a graph is composed by a set of vertices and edges connecting them, but they can be enriched with other information, like labels and attributes. Graphs in which vertices (and edges) can be assigned to attributes of some data type are often called attributed graphs. Attributed graphs generally consist of two parts: a graph-part and a data-part. The data-part includes an algebra which defines values and algebraic operations over these values. An algebra is a semantical model of a signature [20]. As an analogy, we can see a signature as the interface of a program and an algebra as the implementation of this program. An algebra homomorphism relates two algebras over the same signature, identifying their values and operations.

Definition 1 [Signature] *A* signature $\Sigma = (S, OP)$ *consists of a set S of sorts and a family $OP = (OP_{w,s})_{(w,s)\in S^*\times S}$ of operation symbols. For an operation $op \in OP_{w,s}$, we can write $op : w \to s$ or $op : s_0, \ldots, s_n \to s$, where $w = s_0, \ldots, s_n$. If $w = \varepsilon$, then the operation $op :\to s$ is called constant.*

Definition 2 [Algebra and homomorphism] *Given a signature $\Sigma = (S, OP)$, a $\Sigma-$*algebra *$A = (\{A_s | s \in S\}, \{op_A | op \in OP\})$ is defined by:*

- *a carrier set A_s for each sort $s \in S$;*
- *a constant $c_A \in A_s$ for each constant $c :\to s \in OP$;*
- *a function $op_A : A_{s_0} \times \cdots \times A_{s_n} \to A_s$ for each operation $op : s_0, \ldots, s_n \to s$.*

The set obtained by the disjoint union of all carrier sets of a Σ-algebra A is denoted by $\mathcal{U}(A)$, i.e., $\mathcal{U}(A) = \biguplus_{s\in S} A_s$.

Given two algebras A and A' of the same signature $\Sigma = (S, OP)$ or specification $Spec = (S, OP, E)$, a (partial) homomorphism *$h : A \to A'$, also called Σ* or *$Spec$*-homomorphism *is a family $h = (h_s)_{s\in S}$ of functions $h_s : A_s \to A'_s$ such that:*

- *for each $c :\to s \in OP$, we have $h_s(c_A) = c_{A'}$;*
- *for each $op : s_0, \ldots, s_n \to s \in OP$, we have $h_s(op_A(x_0, \ldots, x_n)) = op_{A'}(h_{s_0}(x_0), \ldots, h_{s_n}(x_n))$, for all $x_i \in A_{s_i}$.*

A homomorphism is total or injective if all functions are total or injective, respectively, and if all functions are bijective, it is an isomorphism.

A graph is defined by sets of vertices and edges. The set of vertices are partitioned in two sets: a set of graph vertices and the set of data vertices. And the set of edges are also partitioned into two sets: the set of graph edges and node attribute edges. The graph vertices and edges define the graphical part of a graph, while the data vertices and node attribute edges define the data structure of this graph. In this approach the edges are directed, therefore the source and target of each edge must be defined. There are two function determining the source and target of graph edges and two functions defining the source and target of node attribute edges. Graph edges have source and target in graph vertices and node attribute edges associate graph vertices to data vertices. In order to relate two graphs, a graph morphism is defined, mapping all elements of one graph into the corresponding elements of the other. This mapping must preserve the source and target of each edge, i.e., if an edge e_1 is mapped to an edge e_2, the source and target vertices of e_1 must be accordingly mapped to the source and target of e_2.

Definition 3 [Graphs and graph morphisms] *A* graph *G is a tuple $(V_G, V_D, E_G, E_{NA}, src_G, tgt_G, src_{NA}, tgt_{NA})$ defined as follows:*

- *V_G and V_D are sets of graph and data vertices, respectively;*
- *E_G is the set of graph edges, which connect graph vertices and E_{NA} is the sets of node attribute edges, which connect graph vertices to data vertices;*
- *$src_G, tgt_G : E_G \rightarrow V_G$ are total functions, defining source and target of graph edges, respectively;*
- *$src_{NA} : E_{NA} \rightarrow V_G$ and $tgt_{NA} : E_{NA} \rightarrow V_D$ are total functions, defining source and target of node attribute edges, respectively;*

Given two graphs $G = (V_G^G, V_D^G, E_G^G, E_{NA}^G, src_G^G, tgt_G^G, src_{NA}^G, tgt_{NA}^G)$ and $H = (V_G^H, V_D^H, E_G^H, E_{NA}^H, src_G^H, tgt_G^H, src_{NA}^H, tgt_{NA}^H)$, a (partial) graph morphism *$f : G \rightarrow H$ is a tuple $(f_{V_G}, f_{V_D}, f_{E_G}, f_{E_{NA}})$ such that f commutes with all source and target functions, for example $f_{V_G} \circ src_G^G = src_G^H \circ f_{E_G}$. A graph morphism is said to be total or injective if all its components are total or injective functions, respectively.*

An attributed graph is a graph combined with a data algebra over a signature Σ. In the signature, a set of attribute value sorts is established and the corresponding carrier sets are used to define the data vertices. The relation between two attributed graphs are determined by a graph morphism and an algebra homomorphism, which must preserves the mapping between the data vertices.

Definition 4 [Attributed graphs and attributed graph morphisms] *Given a signature Σ, called data signature, an* attributed graph *is a pair $AG = (G, A)$, where A is a $\Sigma-$algebra, called data algebra and G is a graph, such that $V_D = \mathcal{U}(A)$.*

Given two attributed graphs $AG^1 = (G^1, A^1)$ and $AG^2 = (G^2, A^2)$, a (partial) attributed graph morphism *$f : AG^1 \rightarrow AG^2$ is a pair (f_G, f_D), with a graph morphism $f_G : G^1 \rightarrow G^2$ and an algebra homomorphism $f_D : A^1 \rightarrow A^2$, such that the following diagram commutes for all $s \in S$.*

$$\begin{array}{ccc} A_s^1 & \xrightarrow{f_{D_s}} & A_s^2 \\ \downarrow & & \downarrow \\ V_D^1 & \xrightarrow[f_{G_{V_D}}]{} & V_D^2 \end{array}$$

An attributed graph morphism f is said to be total or injective if f_G and f_D are total or injective, respectively. Moreover, f is a monomorphism if f_G is injective and f_D is an isomorphism of $\Sigma-$algebras.

Attributed graphs combined with the concept of typing leads to the notion of typed attributed graphs. For typing the attributed graphs, a type graph over the final Σ-algebra is used. The typing is given by an attributed graph morphism associating each element of a graph to elements of the type graph. Two attributed graphs typed over the same type graph can be related by an attributed graph morphism, which must preserve the types of each graph element.

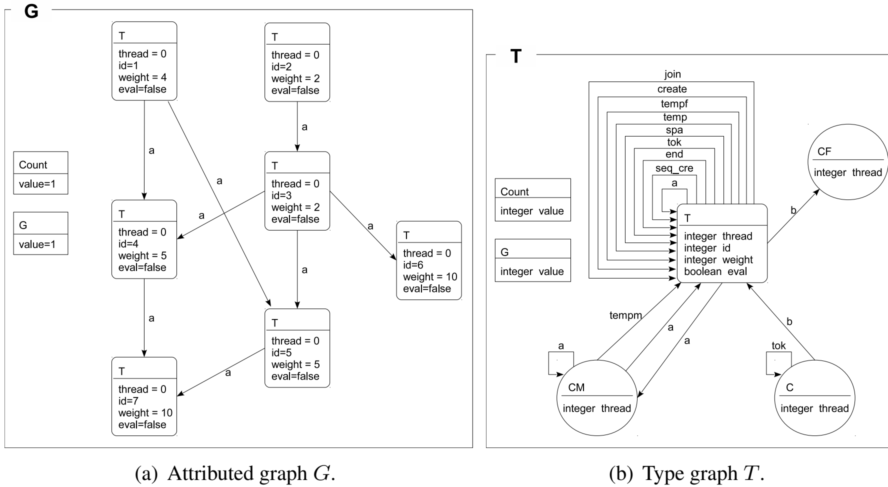
Definition 5 [Typed attributed graphs and typed attributed graph morphisms] *Given a signature $\Sigma = (S, OP)$, an* attributed type graph *is an attributed graph $TG = (T, A)$, where A is the final $\Sigma-$algebra (where all carrier sets of A are singletons). A* typed attributed graph *(AG, t) over TG consists of an attributed graph AG and a total attributed graph morphism $t : AG \rightarrow TG$, called typing morphism.*

Given two typed attributed graphs (AG^1, t^1) and (AG^2, t^2), typed over TG, a (partial) typed attributed graph morphism *$f : (AG^1, t^1) \rightarrow (AG^2, t^2)$ is a (partial) attributed graph morphism $f : AG^1 \rightarrow AG^2$, such that $t^2 \circ f = t^1$. A typed attributed graph morphism $f : (AG^1, t^1) \rightarrow (AG^2, t^2)$ is said to be total, injective or a monomorphism if $f : AG^1 \rightarrow AG^2$ is total, injective or a monomorphism, respectively.*

Typed attributed graphs over an attributed type graph TG and typed attributed graph morphisms form the category $\mathbf{AGraphs_{TG}}$ [20].

Example 1 [Typed attributed graphs] *A typed attributed graph is shown in Figure 1. Vertices are depicted as rectangles or circles, which are divided into two parts, and edges are shaped as arrows connecting their source and target vertices. This graph is attributed over an algebra of integer and boolean. The data vertices and the node attribute edges associating them to graph vertices are inscribed in the bottom part of rectangles or circles. For example, the vertex* T *(on top left of Figure 1(a)) has an attribute named* eval*, whose value is* false*, that is, there is a node attribute edge* eval *connecting the graph vertex* T *to the data vertex* false*. The type graph is depicted in Figure 1(b) and the typing information of G is given by the labels (*T*,* Count *and* G*) in the top part of rectangles or circles and the labels* a *on the arrows.*

A production defines the transformation from a graph to another, identifying which elements should be preserved, consumed or created. In this work we use the double pushout approach (DPO), where a production is defined by two total typed attributed graph morphisms



(a) Attributed graph G. (b) Type graph T.

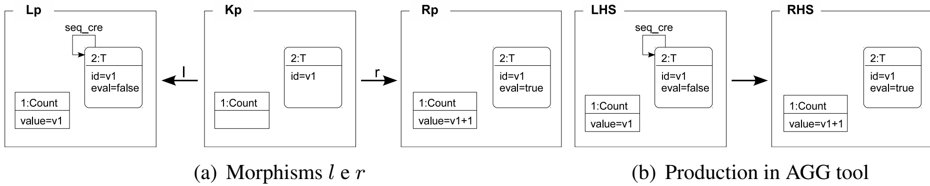
Figure 1. Example of attributed graph typed over T.

$l : K_p \rightarrow L_p$ and $r : K_p \rightarrow R_p$, one mapping elements from the interface (K_p) to the left-hand side (L_p) and another mapping elements from the interface to the right-hand side (R_p). L_p defines the elements that must be present in the graph for the production to be applied; R_p defines the result of application of the production; and K_p defines the context of the production application, i.e., elements that must be in the graph but are not deleted by application of the production. Elements of L_p which are not in the co-domain of l must be deleted, and elements in R_p which are not in the co-domain of r must be created. All graphs of p are typed over the same type graph T, with respect to a signature Σ, and the algebra associated to these graphs is the Σ-termalgebra [20] with the variables X used p.

Definition 6 [Typed attributed graph productions] *Given an attributed type graph TG with data signature Σ, a* (typed attributed) graph production or rule $(p : L_p \stackrel{l}{\leftarrow} K_p \stackrel{r}{\rightarrow} R_p)$ *consists of three typed attributed graphs L_p (left-hand side), K_p (context) and R_p (right-hand side), with a common $\Sigma-$algebra $T_\Sigma(X)$ (the $\Sigma-$termalgebra with variables X); a typed attributed graph monomorphism l; and a typed attributed graph morphism r.*

Example 2 [Graph production] *An example of production is shown in Figure 2(a). The morphisms l and r are defined by the numbers associated to each element of the graphs. The required elements to apply this production are vertices* T *and* Count *and edge* seq_cre *in graph L_p, where the attribute* eval *is* false *and the attributes* value *and* id *are associated the same value $v1$. Among these elements, the vertices* T *and* Count *are preserved and the other*

ones are deleted. The created elements are the node attribute edges associating new values to the attributes in graph R_p, for example the attribute value *is associated to a new value $v1 + 1$ (that is, this attribute is incremented of 1). The AGG tool, which is used to support the analysis of transformation, uses a more compact representation for a production. The AGG representation for the production p is showed in Figure 2(b). The graph K_p is omitted, but it is determined by the numbered elements. Elements in the left-hand side (LHS) or right-hand side (RHS) that have no associated number are deleted or created by the production, respectively.*



(a) Morphisms l e r (b) Production in AGG tool

Figure 2. Production p.

The application of a production to a graph G is enabled if all elements in its left-hand side can be found in G , i.e., the left-hand side matches with a part of G. A match is defined as a total (typed attributed) graph morphism from the left-hand side of a production to a graph. It is total to ensure the presence of all needed elements in G. In this approach, if there is some edge connected to a deleted vertex or if there are two identified vertices, where one is preserved and the other is deleted, the production cannot be applied.

Definition 7 [Match and gluing conditions] *Given a typed attributed graph production ($p : L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R$), a typed attributed graph G and a total typed attributed graph morphism $m : L \rightarrow G$, with $X = (V_G^X, V_D^X, E_G^X, E_{NA}^X, src_G^X, tgt_G^X, src_{NA}^X, tgt_{NA}^X, D^X, t^X)$ for all $X \in \{L, K, R, G\}$, we can state the following definitions:*

- *the* identification points $IP = \{v \in V_G^L | \exists v' \in V_G^L, v \neq v', m_{V_G}(v) = m_{V_G}(v')\} \cup \{e \in E_i^L | \exists e' \in E_i^L, e \neq e', m_{E_i}(e) = m_{E_i}(e')\}$, *for all $i \in \{G, NA\}$ are graph elements in L that are identified by m;*

- *the* dangling points $DP = \{v \in V_G^L | (\exists e \in (E_G^G - m_{E_G}(E_G^L)), m_{E_G}(v) = src_G^G(e)$ *or* $m_{E_G}(v) = tgt_G^G(e)) \vee (\exists e \in (E_{NA}^G - m_{E_{NA}}(E_{NA}^L)), m_{E_{NA}}(v) = src_{NA}^G(e)\}$ *are graph vertices in L, whose image in G are source or target of an edge that are not in image of m.*

p and m satisfy the gluing condition if all identification and dangling points are elements preserved by p, i.e. they are in $l(K)$. In this case, m is called match *for p at G.*

In this work we consider injective matches, i.e., a match cannot identify distinct elements. Because of this, the identification condition is always satisfied, so it remains to verify the dangling condition in order to apply a production.

The production application, or derivation, is defined as a double pushout in the category $\mathbf{AGraphs_{TG}}$ [20]. Intuitively, G can be transformed by removing the part matched by the production's left-hand side and adding its right-hand side.

Definition 8 [Typed attributed graph transformation] *Given a graph production $(p : L_p \stackrel{l}{\leftarrow} K_p \stackrel{r}{\rightarrow} R_p)$, a typed attributed graph G and a match $m : L_p \rightarrow G$ for p at G. A* direct (typed attributed) graph transformation *from G to the typed attributed graph H (with p at m), denoted by $G \stackrel{p,m}{\Rightarrow} H$, is given by the following double pushout (DPO) diagram in the category $\mathbf{AGraphs_{TG}}$, where (1) and (2) are pushouts:*

$$\begin{array}{ccccc} L_q & \xleftarrow{l} & K_q & \xrightarrow{r} & R_q \\ \downarrow m & (1) & \downarrow k & (2) & \downarrow n \\ G & \xleftarrow{f} & D & \xrightarrow{g} & H \end{array}$$

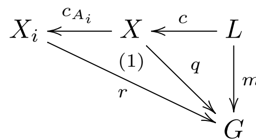
A (typed attributed) graph transformation from G_0 to G_k, *denoted by $G_0 \Rightarrow^* G_k$, is a sequence of direct graph transformations $G_0 \stackrel{p_1,m_1}{\Rightarrow} \cdots \stackrel{p_k,m_k}{\Rightarrow} G_k$.*

The following application conditions are presented as defined in [21]. An application condition establishes structures that are required to apply a production, for example specific conditions on attributes of graphical elements. An application condition for a production consists of a set of constraints, which specifies graph and attribute conditions that have to be fulfilled by a match in order to apply the production. The main constituent for building constraints are equational constraints. An equational constraint is defined by a typed attributed graph morphism which is injective on graph elements and maps each attribute element into a class of equivalent terms. This morphism allow us to specify equations that must be satisfied by a production application. For example, mapping a term $x > 10$ to the quotient term $[true]$ we are defining the equation $x > 10 = true$, or simply $x > 10$.

Definition 9 [Equational constraint] *Given a typed attributed graph $L = (G, A, t)$ and a congruence relation Θ on A, an* equational constraint *over L is any typed attributed graph morphism $c : L \rightarrow X$, where c_G is injective and c_D is the natural homomorphism from algebra A to the quotient algebra $A_{/\Theta}$.*

An application condition (AC) over a graph L is a set of constraints over L. A constraint is composed by an equational constraint $c : L \to X$ over L and a family of equational constraints $c_{X_i} : X \to X_i$ over X. An application condition restricts the application of a production with respect to a match, i.e., the match must satisfy all constrains in the AC in order to apply the production. A match m for a production $p : L \leftarrow K \to R$ in G satisfies a constraint if for all morphism q from X to G, respecting m (that is, all elements related by c_L must be identified in G by q and m), there are a morphism from X_i to G, respecting q. For the attribute part, the satisfaction of an AC means that all established equations are satisfied by the assignment defined by m, for all variables in L.

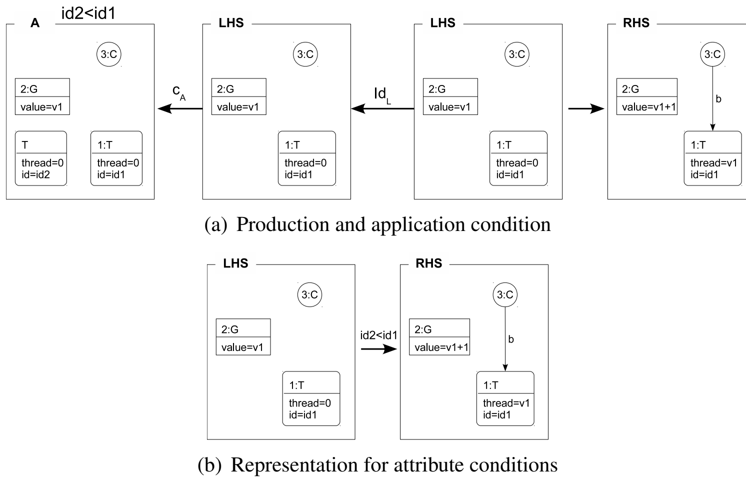
Definition 10 [Application condition (AC)] *Given an attributed type graph TG over a data signature $\Sigma = (S, OP)$ and a set of variables $X = (X_s)_{s \in S}$. Let $L = (G, T_\Sigma(X), t)$ be an attributed graph typed over TG, where $T_\Sigma(X)$ is the $\Sigma-$termalgebra with variables X. A* constraint *$c_L = (c : L \to X, (c_{X_i} : X \to X_i)_{i \in I})$ over L is defined by an equational constraint c over L and an I-indexed set of equational constraints $(c_{X_i})_{i \in I}$ over X. I is finite and possibly empty. A typed attributed graph morphism $m : L \to G$ satisfies a constraint c_L, denoted by $m \vDash_L c_L$, if for all typed attributed graph morphism $q : X \to G$ with $m = q \circ c$, there is an $i \in I$ and a typed attributed graph morphism $r : X_i \to G$, such that the diagram (1) below commutes:*



An application condition *$\mathcal{A}$ over L is a finite set of constraints over L. A typed attributed graph morphism m satisfies an application condition $\mathcal{A}$ if $m \vDash_L c_L$ for all $c_L \in \mathcal{A}$.*

Example 3 [Application condition] *Figure 3(a) shows a production and an application condition having only one constraint $c_L : (id_L, c_A)$. id_L is the identity morphism of L. The morphism c_A is identical on graph elements, and on attribute elements it corresponds to equation $id2 < id1$ (depicted over graph A). This application condition defines an attribute condition, that is, this production can only be applied if the equation $id2 < id1$ is satisfied by the variable assignment defined by the match. If the application condition defines only attribute conditions, we can write the corresponding equations over the production arrow (see Figure 3(b)).*

It is also possible to define negative application condition to restrict the production applications. A negative application condition defines forbidden elements for a production application. It is also defined by a set of morphisms (negative constraints) from the left-hand



(a) Production and application condition

(b) Representation for attribute conditions

Figure 3. Example of application condition.

side of the production. In contrast to AC a match $m : L \to G$ satisfies a negative constraint $n : L \to N$ if there is no morphism from $N \to G$ respecting m.

Definition 11 [Negative application condition (NAC)] *Given an attributed type graph TG, a* negative application condition *over a typed (over TG) attributed graph L is a finite set $\mathcal{N}$ of total typed attributed graph morphisms $n : L \to N$, called* negative constraints*, where N is an attributed graph typed over TG.*

Given a total typed attributed graph morphism $m : L \to G$, m satisfies a negative constraint $n : L \to N \in \mathcal{N}$ if there is no total typed attributed graph morphism $n' : N \to G$, such that $m = n' \circ n$. m satisfies a negative application condition $\mathcal{N}$ if it satisfies all negative constraints $n \in \mathcal{N}$.

A conditional production is defined by a production, an application condition and a negative application condition. A typed attributed graph grammar GG consists of signature Σ, which defines the data values and operations used in all graphs of GG; a type graph T attributed over the final Σ-algebra; an initial attributed graph typed over T; and a set of conditional productions.

Definition 12 [Conditional productions and typed attributed graph grammar] *A* typed attributed graph production with application and negative application conditions*, or simply*

conditional production or conditional rule*, is a tuple* $(p, \mathcal{A}(p), \mathcal{N}(p))$ *consisting of a graph production* p*, an application condition* $\mathcal{A}(p)$ *over* L_p*, and a negative application condition* $\mathcal{N}(p)$ *over* L_p*.*

A (typed attributed) graph grammar $GG = (\Sigma, TG, G_0, P)$ *consists of a data signature* Σ*, an attributed type graph* TG *with data signature* Σ*, an initial attributed graph* G_0 *typed over* TG *and a set of conditional productions* P *typed over* TG*.*

Example 4 [Conditional production] *A conditional production is depicted in Figure 4. It is the same production of Example 2 adding an AC and a NAC. The equation* $t1! = 0$ *defined by the AC is depicted over the production arrow. The NAC is defined by just one negative constraint* $n : L \to NAC1$*. The forbidden elements are those in* $NAC1$ *that do not have any associated number, that is: the vertex* T*, which does not have an associated number; its attribute* thread*, with value* 0*; and the edge connecting the vertex* 2:T *to* T*. This production can be applied to a graph* G *if: all required elements can be found in* G*; no forbidden element can be found in* G*; and the attribute* thread *of* 2:T *is not associated to* 0 *in* G*.*

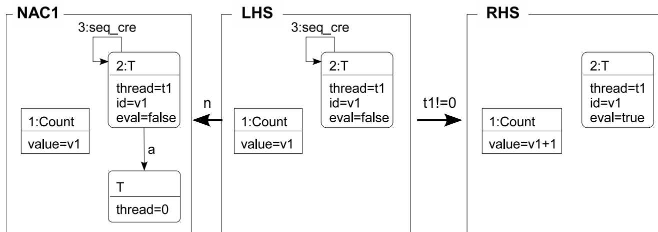


Figure 4. Conditional production p.

A direct graph transformation with ACs and NACs is a direct graph transformation where the match satisfies the AC and NAC of the applied production.

Definition 13 [Typed attributed graph transformation with ACs and NACs] *Given a conditional production* $(p, \mathcal{A}(p), \mathcal{N}(p))$ *and a match* m *for* p *in a typed attributed graph* G*, satisfying* $\mathcal{A}(p)$ *and* $\mathcal{N}(p)$*, a* direct (typed attributed) graph transformation with AC and NAC *from* G *with* p *at* m *is the direct graph transformation* $G \overset{p,m}{\Rightarrow} H$*. A* (typed attributed) graph transformation with ACs and NACs *from* G_0 *to* G_k *is a sequence of direct graph transformation with AC and NAC* $G_0 \overset{p_1,m_1}{\Rightarrow} \cdots \overset{p_k,m_k}{\Rightarrow} G_k$*.*

Example 5 [Direct graph transformation] *Figure 5 shows a direct derivation from* G *to* H *with the conditional production* p *(see Figure 4). There is a match* m *of* LHS *in* G*, which*

is highlighted in the figure. Beyond the existence of m, we can easily see that m satisfies the AC and the NAC of p, i.e., the value of attribute thread *of vertex* 2:T *is not* 0*, and there is no vertex* T*, with attribute* thread $= 0$*, connected to* 2:T *by an edge* a*. In this case, G can be transformed into H excluding the* seq_cre *edge and changing the values of attributes* value *(of vertex* Count*) and* eval *(of vertex* 2:T*) to* 3 *($v1 + 1$) and to* true*, respectively.*

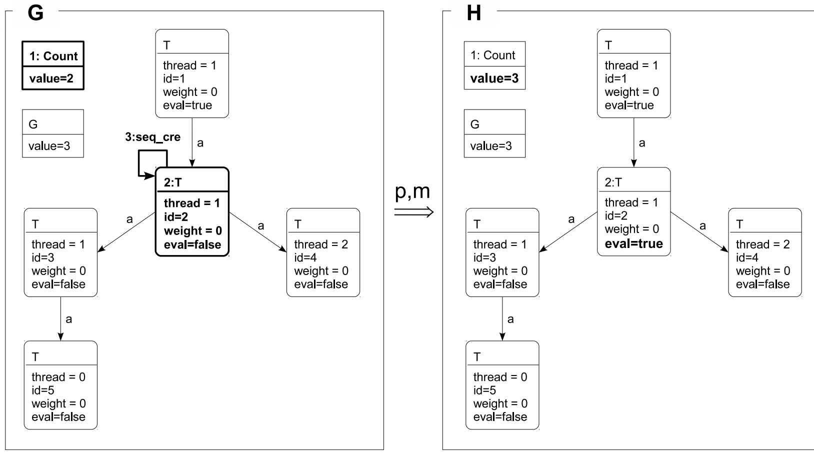


Figure 5. Direct graph transformation from G to H with production p in Figure 4.

3 Transforming DAGs into DCGs

Canonical scheduling algorithms found in literature [22, 23, 24] consider parallel programs modeled as a DAG of tasks. List scheduling [4, 25] is a class of algorithms that offer a proven efficiency when the reveive a static DAG as input. Scheduling algorithms for multithread environments, in turn, deal with DCGs, cyclic graphs built at runtime, whose scheduling unit is the thread. Nevertheless, we can adapt list algorithms to work in multithread environments, like many current tools do [13, 26, 11]. In order to compare directly the performance of scheduling algorithms that work in different scenarios, we developed a translation process to transform DAGs into DCGs, both representing equivalent programs.

3.1 Graphs used to express parallelism

DCGs (Directed Cyclic Graphs) are graphs used to describe multithreaded programs. DAGs (Directed Acyclic Graphs), in turn, are graphs, frequently used to describe parallel

programs in the dataflow model. Both of them have weighted vertices and edges, but the meaning of these elements differ from one to the other. Weights for vertices represent processing cost and weights for edges describe data communication costs.

DAGs' vertices represent computations that process input data and produce output data that can serve as input for other vertices. In this context, an edge defines a dataflow between two tasks. DCGs' vertices, on the other hand, represent containers of tasks, that we call here as threads, and the edges represent operations of creation and synchronization of threads.

We noticed that certain arrangements of tasks show the same structure and semantics as create and join operations in multithreaded programs. In this work these structures have been mapped to segments of a DCG. The resulting DCGs represent valid multithread programs that use only *create* and *join* primitives that can manipulate multiple threads at once and generate unstructured graphs, like the ones found in the Anahy [13] environment.

Formally, a DAG is a graph without cycles. A DCG is a graph without loop edges that can have cycles. Besides, the set of edges in a DCG is partitioned into two subsets, one to model *creates* and the other one *joins*.

Definition 14 [Directed Acyclic Graph and Directed Cyclic Graph] *A* directed acyclic graph (DAG) *is a tuple $G = (V, E, src, tgt)$ where V is a set of vertices, E is a set of edges, $src, tgt : E \to V$ are total functions, defining source and target of edges, respectively, such that for all $e \in E, src(e) \neq tgt(e)$ and for each $p = e_1 e_2 \cdots e_n \in E^*$, if $tgt(e_i) = src(e_{i+1})$, with $i \in \{1, \ldots, n-1\}$, then $src(e_1) \neq tgt(e_n)$.*

A directed cyclic graph (DCG) *is a tuple $G = (V, E_c, E_j, src, tgt)$ where V is a set of vertices, E_c is a set of* create *edges, E_j is a set of* join *edges, $src, tgt : E_c \uplus E_j \to V$ are total functions, defining source and target of edges, respectively, such that $\forall e \in E_c \uplus E_j, src(e) \neq tgt(e)$. In what follows $E_c \uplus E_j$ is denoted by E.*

Vertices and edges can have associated weights, denoted by $|v|$ and $|e|$, respectively, where $v \in V$ and $e \in E$.

Example 6 [Directed Acyclic Graph and Directed Cyclic Graph] *Tasks and dependencies among them can be modeled by a DAG. Figure 1(a) shows an example of a DAG describing a parallel program with seven tasks whose dependencies are represented by edges labeled with a.*

Threads and create-join relations can be represented by a DCG. Figure 20(b) illustrates an example of a DCG modeling a multithreaded program with four threads whose create-join relations are described by edges.

The DCG represented in Figure 20(b) is the graph obtained from the DAG illustrated in Figure 1(a) using the graph transformation rules presented in the next section.

3.2 Graph Grammar Specification

Roughly speaking, the creation of a DCG $C = (V^C, E_c^C, E_j^C, src^C, tgt^C)$ representing a DAG $G = (V^G, E^G, src^G, tgt^G)$ preserves the original structure of G. The creation process consists of encapsulating a sequence of $n \geq 1$ tasks of the original G in the context of C threads. The directed edges between tasks in G are preserved in C, but only the edges representing dependencies between tasks of different threads are included in E^C. Those edges represent the creation or the synchronization of threads.

In order to translate a DAG into a DCG, a (typed attributed) graph transformation is defined. The rules used in such graph transformation are those defined in Subsection 3.2.1. The full process of transformation is performed in two phases. The first one distributes the tasks in threads, and the second one constructs the DCG abstracting the internal tasks of each thread.

In order to proceed a concrete translation an initial graph must be defined. Graph transformation rules enriched with an initial graph defines a (typed attributed) graph grammar. The initial graph changes according to the application, although the type graph remains the same. According to the type graph T, depicted in Figure 1(b), a task is defined with 4 different attributes, one evaluation attribute (eval) of type *Bool* to express if the task was fully evaluated and three other attributes of type *Int*: an identifier (id), a weight (weight) corresponding to the computational cost associated to the task and a thread attribute (thread) indicating the number of the thread the task is included (this attribute is set to zero when the task is not included in any thread).

Circles of type C, CM and CF are tasks called *conforming tasks*. A conforming task can be considered as an auxiliary task with no computational cost, which serves to rearrange a certain configuration of tasks only allowed in DAGs to fit DGCs' rules. The adopted notation C, CM and CF is used to differentiate the position of the conforming task, indicating, respectively, if it is created in the beginning (as the first task), in the middle or in the end (as the last task) of a thread.

Additionally, rectangles of type Count and G, both with an attribute value of type *Int* initialized with 1, are used as global variables. Node G represents the number of the next thread to be created and node Count is a counter used to control the number of the task to be evaluated.

For instance, graph G illustrated in Figure 1(a) is a possible initial state. It contains seven tasks, which are connected by edges of type a, describing the dependencies between

them. All tasks have attribute thread initialized with zero and the attribute eval initialized with false. Each task has a different identifier and a specific weight. Both global variables are setted to one. In this work, we consider that in all state graphs, any vertex of type C, CM or CF has weight zero. For this reason, we omit the attribute weight for such vertices.

3.2.1 Graph Transformation Rules

Figures 6 to 15 present a set of graph transformation rules that specifies the creation of a DCG from a DAG. Figures 6 to 12 describe the first phase of the transformation and Figures 13 to 15 the second one. Rules are specified with a priority order. This feature, though not very common in graph grammar specifications, is available in the AGG tool set [17], which was used to edit and simulate the proposed translation. Rule priorities provide a way to schedule the application of rules: as long as a high-priority rule is enabled, no lower-priority rules can be scheduled for application. In general, this strategy simplifies the rules specification, avoiding the creation of extra components (flags) necessary to enforce the order of rule applications.

The rule with the highest priority, named `bunch`, showed in Figure 6, determines the junction of two tasks, 1:T and 2:T, when 1:T is an immediate predecessor of 2:T (that is, there is an edge e of type a such that $src(e)$ = 1:T and $tgt(e)$ = 2:T), but neither 1:T is an immediate predecessor of any other task (determined by NAC2 and NAC4) or 2:T has another immediate predecessor (specified by NAC1 and NAC3). That is, there is no edge e of type a such that $src(e) =$ 1:T and $tgt(e) \neq$ 2:T and there is no edge e of type a such that $tgt(e) =$ 2:T and $src(e) \neq$ 1:T. The cost of the resulting task 1,2:T is |1:T | + |2:T | (specified by the weight attribute).

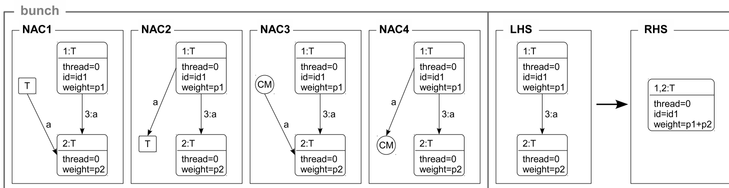


Figure 6. Rule Bunch with Priority 1.

Rules `beginC` and `beginT` are depicted in Figure 7. These rules identify tasks in G that have no predecessors. The result of the application of these rules is that each task with no precedence constraints will represent the first task to be executed in the context of a thread in C. Rule `beginC` creates a conforming task to represent the beginning of execution with (possibly) multiple thread creation. Notice that a conforming task C is introduced just once (according to NAC1), in the first created thread, including the task with the lowest

identifier. Multiple applications of rule `beginT` specify a multiple creation of threads, each one containing a beginning task (without predecessors) of the given DAG. The sequence of threads creation is determined by the lowest identifier of the beginning tasks that are note included in any thread. The node of type G represents a global variable containing the number of the next thread to be created. Conversely, rule `end` (Figure 7) identifies tasks without successors in G and sets them as evaluated.

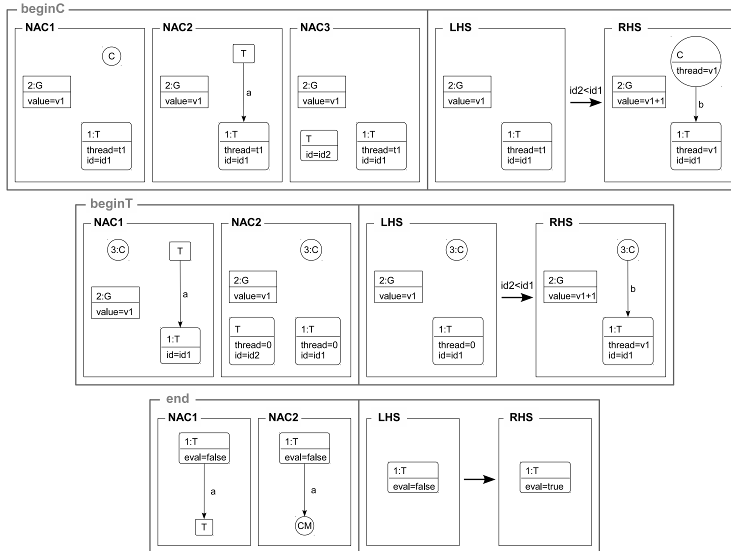


Figure 7. Rules for *thread* beginnings and last tasks identification with Priority 2.

Next rules specify the following patterns: *create*, *join*, *broadcast* and *spawn*. The general idea is to explore tasks from lower values (identifiers) until the greatest one, respecting the priority order. That is, the task to be evaluated will be the task still not evaluated (with attribute eval false) with id equals to the counter that matches with the rule of highest priority. In case that there is no task with eval false with id equals to the counter, rule `count` (depicted in Figure 12) is applied, incrementing the counter. In order to guarantee that this rule will not be active after the evaluation of the last task, its application conditions also require the existence of a task with id greater than the counter. The patterns *create* and *join* correspond respectively to the creation and synchronization of threads.

The pattern *create* is defined by rules `create` and `createSeq` (Figure 8). Rule `create` enrols a task (not included in any thread) in a thread. That is, when a task still not evaluated (with eval = false) has no successor in the same thread, its immediate successor with

the lowest identifier (not included in any thread) is included in its thread context. If the task in evaluation (the task with the lower identifier with eval = false) already has a successor in the same thread, but has also one immediate successor that is not in any thread, rule `createSeq` is applied, and a new thread is added to the DCG. At last, when all immediate successors of the task in evaluation already be in the context of a thread, rule `createEnd` (Figure 8) is applied, setting the task as evaluated.

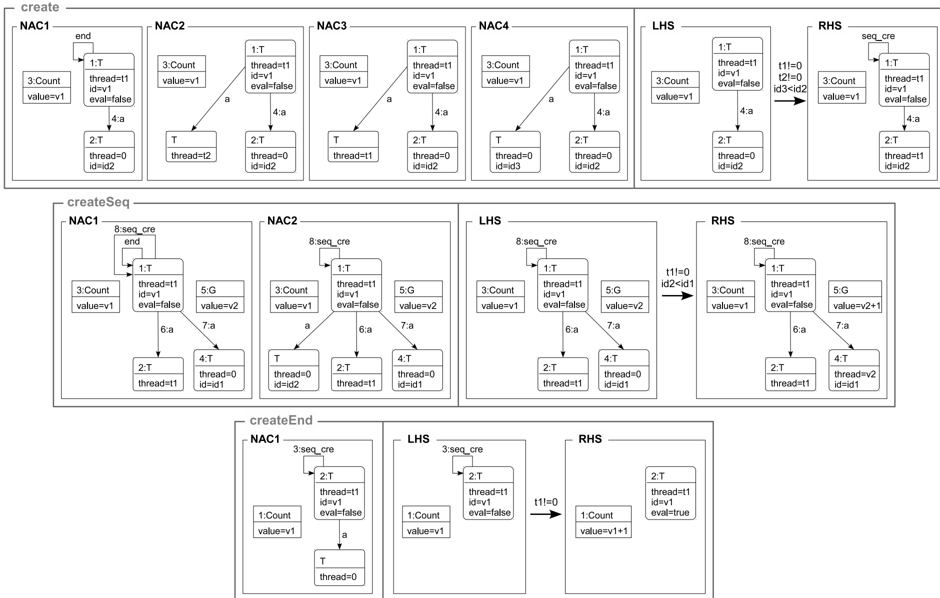


Figure 8. Rules for the pattern *create* with Priority 3.

The pattern *join* identifies the termination of threads and the respective synchronization. Particularly, rules `join` and `joinCM`, showed in Figure 9, are applied when a task still not evaluated is included in a thread with an immediate successor in another thread, identifying the termination of its thread and a synchronization point. The task in evaluation is labeled with end, indicating that its thread must be closed, and the synchronization point is labeled with tok. Rule `join` is applied when the antecedent task of the synchronization task is a T task and `joinCM` is applied when it is a CM task.

Despite the end of a thread being identified, the last task executed by this thread may have other dependencies matching *broadcast* and/or *spawn* patterns. A broadcast corresponds to the synchronization of the end of the current thread with other threads. Rule `broadcast`, depicted in Figure 10, identifies successors of the task labeled with end still

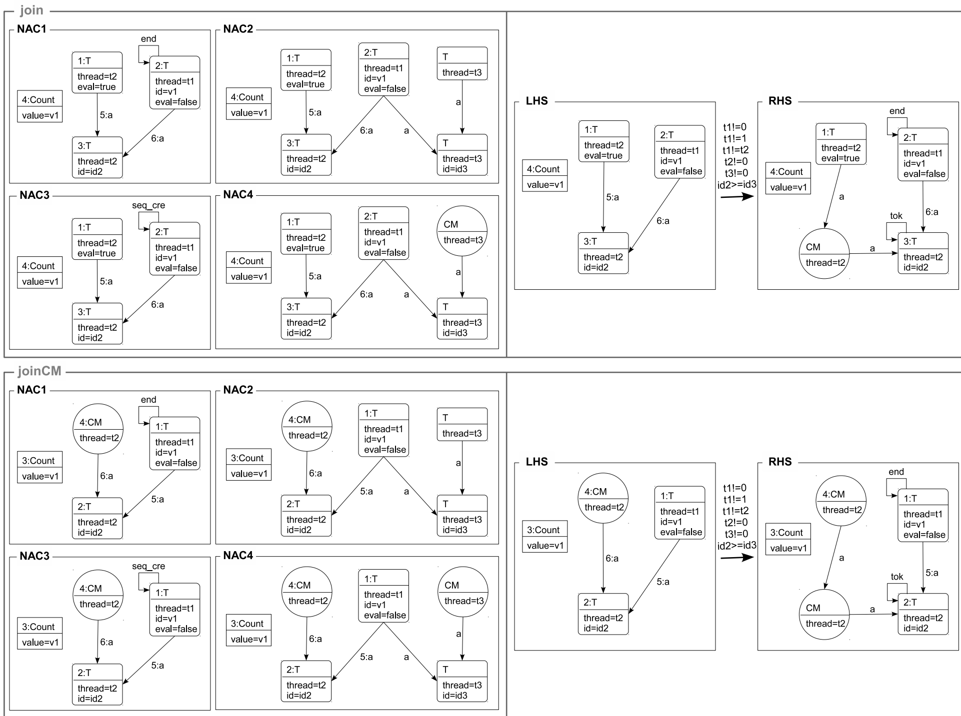


Figure 9. Rules for pattern the *join* with Priority 3.

not evaluated which are in threads, and creates a conforming task CM preceding these tasks. Rule `broadcastSeq` (Figure 10) adjusts the dependencies of these tasks to the created conforming tasks. Rule `broadcastCM` and `broadcastSeqCM` have respectively the same meaning of `broadcast` and `broadcastSeq`, but now considering that the successors of the task labeled with end are conforming tasks CM. The latter two rules are also shown in Figure 10.

A *spawn* corresponds to creation of new threads. Rules `spawn` and `spawnSeq`, illustrated in Figure 11, identify immediate successors of the task labeled with end (task in evaluation) that are not in any thread, include them in new threads, and label them with spa. Rule `spawn` also creates a conforming task CM in the same thread of the task in evaluation to express already evaluated dependencies. Since this conforming task is created just once (specified in the negative application condition of `spawn`), first rule `spawn` must be applied. After the creation of the conforming task CM, just rule `spawnSeq` can be applied. The

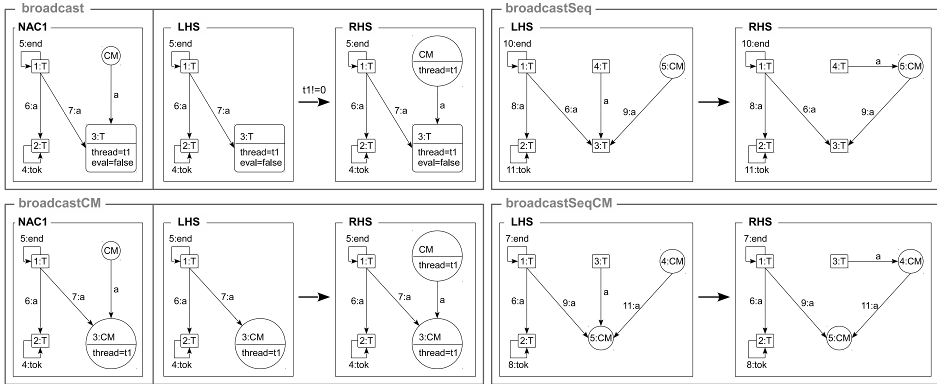


Figure 10. Rules for the pattern *broadcast* with Priority 3.

current dependencies from the task in evaluation to tasks in other threads is adjusted to the created conforming task in rules `spawn` and `spawnAux` (Figure 11).

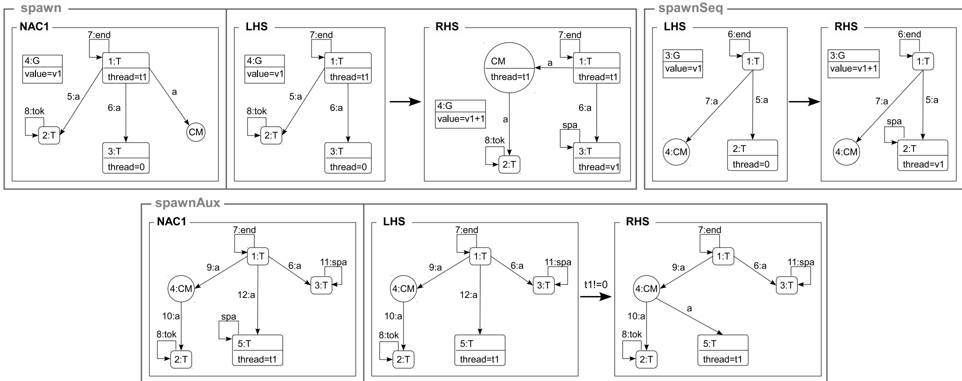


Figure 11. Rules for the pattern *spawn* with Priority 4.

After identifying all broadcasts and/or spawns to the task in evaluation, rules `conclude` or `concludeCM` (Figure 12) can be matched. Rule `conclude` is applied when the task labeled with end has a successor task of type T and `concludeCM` is applied whether it has a successor task of type CM . Both set the task labeled with end as evaluated and delete the flags end and tok.

Finally, rules with lowest priority `endC` and `endT` of the first phase, depicted in

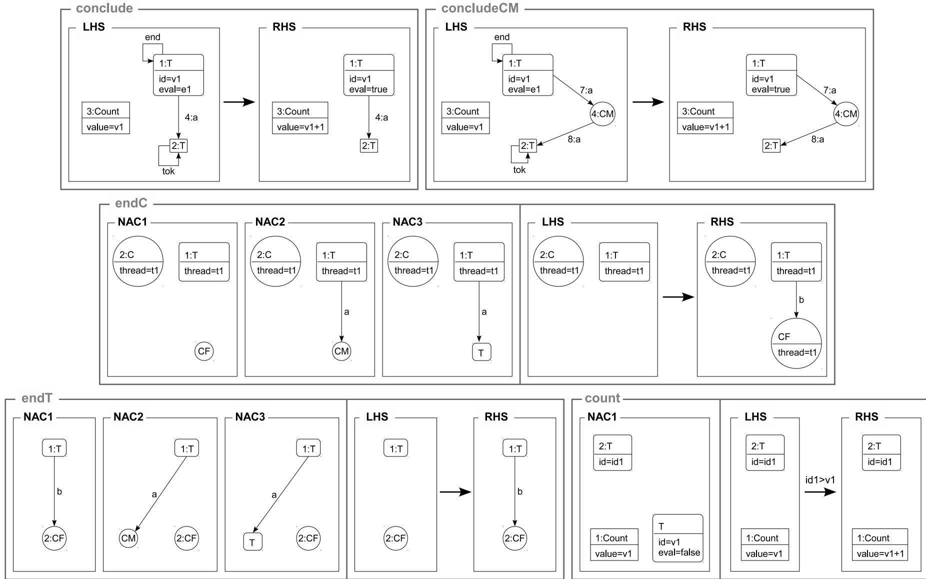


Figure 12. Rules for conclusion with Priority 5, *thread* endings with Priority 6 and counter evolution with Priority 3.

Figure 12, identify tasks without successors and links them to a final conforming task. Particularly, rule `endC` is applied just once, creating the final conforming task in the end of the main thread (thread containing the conforming task C). After that, rule `endT` can be applied.

The graph obtained from the first phase is submitted to a new set of rules, which composes the second phase of the transformation. In this phase, each rule has a different priority level. This set of rules abstracts the internal tasks of a thread, gluing them in a unique vertex. Besides, in this phase the *create* and *join* edges are identified (in the resulting graph of first phase all edges are of type a or b).

Rules `replaceC1`, `replaceC2` and `replaceC3`, depicted in Figure 13, replaces the beginning conforming tasks by useless tasks of type T . This must be done just to allow the gluing of tasks of different types into a unique vertex. Rules `replaceCM1` to `replaceCM6` (see Figure 14) and rules `replaceCF1` to `replaceCF3` (see Figure 13) are responsible for replacing the medium and final conforming tasks by such useless tasks, respectively.

Rule `identifyCreate`, illustrated in Figure 15, identifies which are the edges of type a representing threads creation, and replaces them by create edges. This edges are those

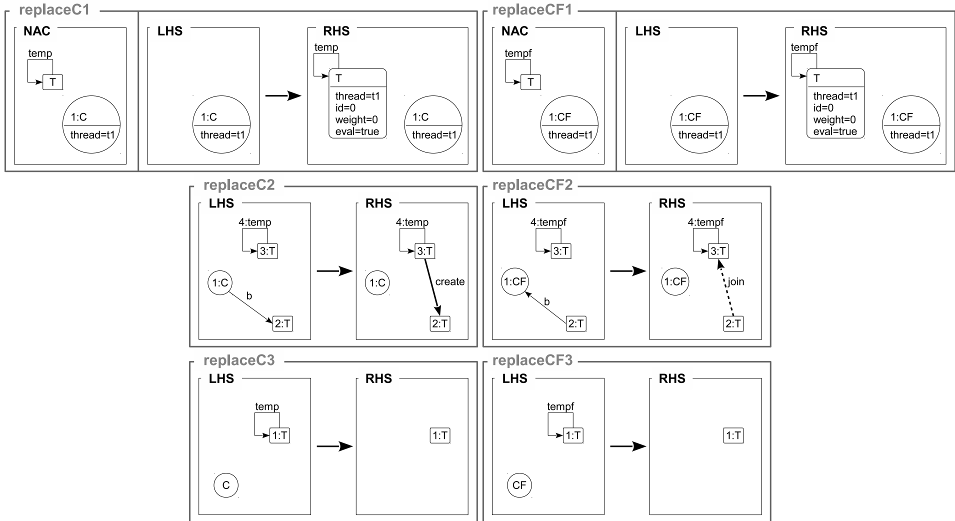


Figure 13. Second phase: rules to replace beginning and final conforming tasks.

between tasks in different threads whose target task does not have antecedent tasks in the same thread. In its turn, rule `identifyJoin` (Figure 15) identifies which are the edges of type a representing threads synchronization, and replaces them by join edges. This edges are those between tasks in different threads whose target task has an antecedent task in the same thread.

Rule `glue`, depicted in Figure 15, collapses all tasks that are in the same thread in a unique vertex, deleting the a edges. The weight attributed to such vertex is given by the sum of weights of each collapsed task.

3.3 Remarks on the transformation process

Once the first phase of the transformation process is finished, the corresponding graph has all information contained in the original DAG, added to the new information about threads and thread operations. In this representation the edges which connect vertices (tasks) with the same `thread` identifier represent the execution order of tasks in that thread. Edges connecting vertices with different `thread` identifiers represent either a *create* operation, when the edge ends on the first task of a thread, or a *join* operation, when the edge begins on the last task of a thread. The DGC obtained after the second phase is a higher level abstraction that considers all tasks in a thread as a unique vertex. Only edges, which connect

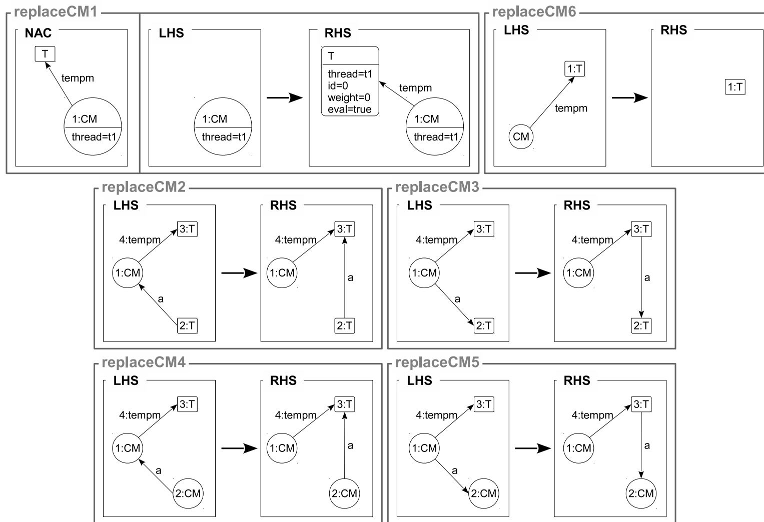


Figure 14. Second phase: rules to replace medium conforming tasks.

tasks in different threads, are considered in this higher level. This simplified abstraction of a multithreaded program is, in fact, the structure manipulated by scheduling algorithms to take decisions during execution time. Anahy [13] uses this light representation of the graph to reduce the scheduling costs and achieves competitive performance levels.

After the transformation process, because of the way rules are applied, there is a tendency of threads closer to root of the DCG to aggregate higher number of tasks. Based on this tendency, a scheduling heuristic can be applied to prioritize the execution of these threads assuming that they are part of the critical path of the application. This is actually the strategy used in the Anahy environment. However, different identifiers distribution for the same DAG structure can result in totally different DCGs, affecting the performance of the scheduling algorithms that take these DCGs as input. Once the programmer knows how the transformation process works, he or she can arrange task identifiers in order to fit the scheduling strategy to be applied on the resulting DCG. In other words, the programmer has to consider the runtime system policies to better arrange the order of creation of tasks. In other hand, if the transformation process did not consider the task identifiers as a priority criteria to apply the rules, the distribution of tasks in the context of threads could be affected. As result, different multithread programs for a same input DAG could be generated resulting in different execution behaviors.

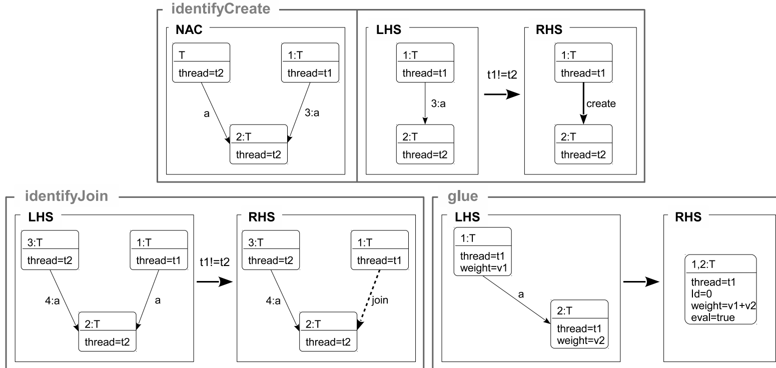


Figure 15. Second phase: rules to identify create and join relations and rule to collapse tasks in the same thread.

4 Transformation Analysis

Besides the automatic translation and precise definition of the mapping, the use of a formal language also allowed the verification of some properties about the translation.

Important features that are required, when graph grammars are used to specify model transformations, are termination and confluence. Only under these conditions the existence and uniqueness of the outcoming model may be guaranteed. Now we discuss why our graph grammar specification for the transformation of DAGs into DCGs satisfies such requirements. The confluence verification was done using the AGG tool [17]. In spite of AGG tool be based on the Single Pushout approach, it allows to simulate the DPO approach selecting the identification and dangling conditions in the transformation settings. We could not use the same tool for termination because it just support such kind of analysis for grammars defined with layered rules. In our case, the grammar had to be defined with rule priorities, which determine that the activated rules with higher priority must be executed first, and after the execution of a rule of lower priority, a rule of a higher group of priority can be active. In case of layered grammar, each group of rules is executed at most once.

Termination The transformation process finishes when there is no rule that can be applied in the current state. For instance, a transformation does not finish when there is a rule that creates new components and is always active (that is, it can always be applied in the state graph). Another situation is when rules can be applied in cycle, that is, components are created by rules that are deleted by others, in a situation where the application of rules that consume the items activate rules that create them.

We assume that the initial state graph must respect the following characteristics: it is

always finite; all tasks must be of type T, initialized with different ids and with attribute eval *false*; and there is a vertex of type Count (counter) with value 1. In what follows, we discuss the termination of the first phase. The finite applicability of each (group of) rule(s) is presented by priorities.

- Rule `bunch` of higher priority collapses tasks of type T that are not in threads and that are linked by an directed edge. The NAC of the rule requires that the source task has no other successor and the target task has no other predecessor. Since there is a finite number of tasks and there is no rule application that generates a graph with the imposed restrictions, rule `bunch` will be applied from the start graph a finite number of times and, after the application of any rule of lower priority, it will never be applied again. This is because only rules that create or delete edges between T tasks could generate a graph in which `bunch` could be applied again. We do not have any rule in the grammar that creates edges between tasks of type T. Rules `join`, `joinCM` and `spawn` delete edges between T tasks that are already in threads (then these tasks cannot be collapsed). Rule `broadcastSeq` that deletes an edge between T tasks, in fact, move the respective edge keeping its source vertex. The target task of the original edge could not be collapsed because it is already in thread. The new target vertex of the moved edge is a CM task (and then cannot be collapsed). Similarly, rule `spawnAux` move an edge between T tasks, keeping its target vertex. Also, the source of the original edge is already in thread and the new source task of the moved edge is a CM task.
- In the second level of priority we have rule `beginC`, which is applied just once and rules `beginT` and `end` that are applied a finite number of times and, after the application of any rule of lower priority, they will never be applied again. Rule `beginC` creates a C component that is prohibited by its NAC. Since it is not deleted by any other rule, after applied, `beginC` will never be applied again. Rule `beginT` identifies tasks without predecessors that are not in threads and include them in threads. Since there is a finite number of tasks and there is no rule to remove tasks from threads, it will be applied a fixed number of times. Similarly, `end` identifies tasks without successors and set them as evaluated. Since there is a finite number of tasks and there is no rule to set attribute eval of a task as *false* it will be applied a limited number of times.
- After the application of rules in the first two group of priorities, just three rules of the next three group of priorities can be active: `create`, `join` or `count`. This is because the remaining rules require components to be applied that are just created after the first application of these rules. In general, the transformation process consists in evaluate the task with id equals to the counter value that has not been evaluated (which has eval *false*). After evaluated, the task is set

as evaluated (eval is set to *true*). There is no rule which change attribute eval from *true* to *false*. Rule `count` is applied when there is no task that has not been evaluated (with eval false) with id equals to the counter. In this case, the counter is incremented. In order to guarantee that this rule could not be applied an illimitable number of times, its application conditions also require a task with id greater than the counter so that the rule could be activated. If there is a task with id equals to the counter still not evaluated, `create` or `join` can be active. Both rules will never be activated simultaneously because they have mutually exclusive application conditions: `create` has a NAC forbidding that the task in evaluation (task with id equals to the counter) has a successor task in thread - application condition of `join`. Rule `create` identifies its successor task with lower identifier that is not in thread and include it in its thread. Once applied, this rule will never be applied to the same task in evaluation (since it create conditions that are forbidden by its NAC). The application of `create` can turn `createSeq` active (it creates a seq_cre edge that is in the application condition of `createSeq`). Rule `createSeq` includes successor tasks of the evaluation task that are not in threads in new threads. Since we have a finite number of successor tasks, this rule is applied a limited number of times (important to notice again that no rule removes tasks from threads). Rule `createEnd` identifies when there is no successor task of the task in evaluation that is not in thread, sets the evaluation task as evaluated (changing the attribute eval from *false* to *true*), deletes the seq_cre edge and increments the counter. Because of that, the task set as evaluated will never be evaluated again.

Rules `join` and `joinCM` are applied at most once for each task in evaluation (task with eval *false* with id equals to the counter value). When applied, the rules create an end edge that is prohibited by its NAC. When such edge is deleted, the task is set as evaluated (and then, the rules could no be applied again for the same task). After the application of one of these rules, broadcasts (rules `broadcast` or `broadcastCM`) or spawns (rule `spawn`) can be identified for the task in evaluation. Rules `broadcast` or `broadcastCM` identify successor tasks of the task labeled with end that are already in thread, creating a predecessor CM task to them. Their NAC forbids the existence of such CM task, assuring the limited number of applications (important to observe that no rule delete CM tasks). The creation of these CM tasks can turn rules `broadcastSeq` and `broadcastSeqCM` active. They move the edges with target in the successor task of the task labeled with end to the CM task created by `broadcast` or `broadcastCM`. Since they delete edges which are required by their application conditions, the number of applications of them is also finite. According to the priorities, spawns are identified after all broadcasts.

- Rule `spawn` identify a successor task of the task labeled with end that is not

in thread, put the task in a new thread and creates a successor CM task to the task labeled with end. Since the created components are prohibited by its NAC, the rule is applied at most once for each task in evaluation. After the `spawn` application, rule `spawnSeq` can be applied (this rule requires components created by `spawn`), recognizing other successor tasks of the task labeled with end that are not in threads and putting them in new threads. The finite number of tasks determines its finite number of applications (note that no rule remove tasks from threads). Rule `spawnAux` move edges with source in the task labeled with end to the CM task created by `spawn`. As `spawnAux` deletes edges that are required for its application, its number of applications is limited. After an spawn no broadcast can be accomplished for the same task in evaluation because an `spawn` application deletes the edge with source in the task labeled with end and target in the task labeled with tok (edge required for a broadcast operation). When no more broadcasts or spawns can be identified rules `conclude` or `concludeCM` are applied, setting the task in evaluation as evaluated and incrementing the counter.

- Finally, we have two rules of lowest priority. Rule `endC` that creates a CF task, which is forbidden by its NAC (and not deleted by any rule), and thus, applied just once. Rule `endT` which is activated by `endC` and creates an edge to the CF task from each task that has no CM or T successors. Due to finite number of tasks and its NAC that prohibit the existence of such edge, the rule is applied a limited number of times.

The termination of the second phase is direct. Rules `replaceC1`, `replaceCM1` and `replaceCF1` are just applied once for each conforming task: they create components that are forbidden by its NACs and when the forbidden elements are deleted, the corresponding conforming task considered in the match is also deleted (and then, they never will be applied for the same conforming task). Since the number of conforming tasks are finite, the application of such group of rules is also finite. Rules `replaceC3`, `replaceCM6` and `replaceCF3` are also just applied once for each conforming task because they delete the corresponding conforming task and there is no rule creating such type of vertex. Rules `replaceC2` and `replaceCF2` delete edges of type b and create edges of type create and join, respectively. Since the number of b edges is finite and there is no rule creating this type of edge, the sequence of these rule applications is finite. Rules `replaceCM2` to `replaceCM5` move a edges with source (or target) in a medium conforming task to a corresponding useless task. Since the number of medium conforming tasks are finite, their incident edges are also finite and there is no rule creating such kind of edges, the applications of these rules terminate. Rule `identifyJoin` (`identifyCreate`) deletes a edges between tasks in different threads whose target task has (does not have, respectively) antecedent tasks in the same thread and creates join (create) edges. Considering that the amount of edges that fits

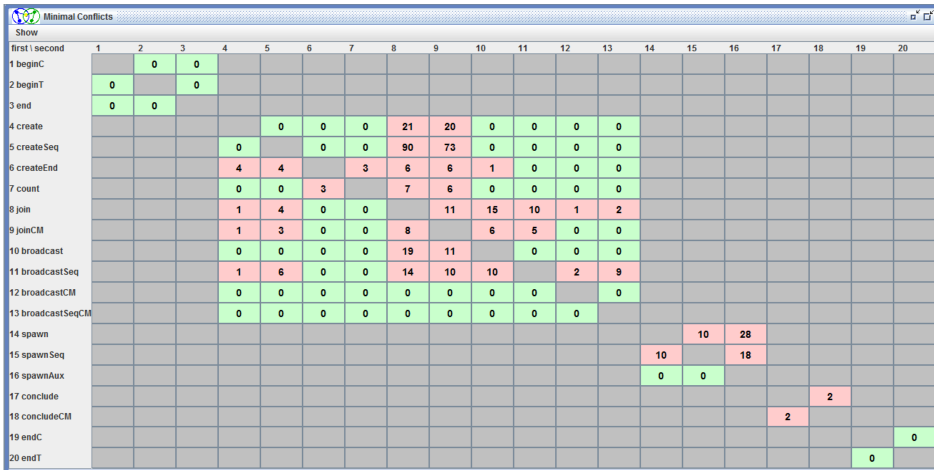
on this situation is finite and rules that create a edges will not be applied again, the number of applications of these rules is also finite. The applications of rule `glue` terminate whereas this rule collapse tasks in the same thread and the number of these tasks is finite. Moreover, there is no rule creating T vertex after the application of the rule `glue`.

Confluence A model transformation is confluent if for each source model the process of transformation results in a unique target model. Critical pair analysis [18] is generally used to check if a transformation is confluent. A critical pair is a pair of transformations both starting at a common graph G such that both transformations are in conflict, and graph G is minimal according to the rules applied (that is, G only contains elements that are in the image of the matches of both rules). There exists a critical pair like above if, and only if, one rule may disable the other one. There are three reasons why rule applications can be conflicting: (i) one rule application deletes a graph component which is in the match of another rule application; (ii) one rule application generates graph components in a way that a graph structure would occur which is prohibited by a NAC of another rule application; (iii) one rule application changes attributes being in the match of another rule application. A graph grammar system is confluent if it is locally confluent and terminates. A system is locally confluent if all critical pairs are confluent, that is, all critical pairs can be derived by a sequence of transformations that leads them to a common successor graph.

We have used the AGG tool [17] to proceed with the critical pair analysis. After computation, the set of critical pairs precisely represents all potential conflicts in the grammar. In order to detect all potential conflicts of type (i) or (iii) described above, for each pair of rules $p1 : L1 \to R1$ and $p2 : L2 \to R2$, AGG computes graph G by overlapping $L1$ and $L2$ in all possible ways, such that the intersection of $L1$ and $L2$ contains at least one item that is deleted or changed by one of the rules and both rules are applicable to G at their respective occurrences. Potential conflicts of type (ii) are found by gluing the right-hand side of the first rule and the left-hand side together with NAC elements of the second rule.

Figure 16 shows the number of potential conflicts between each pair of rules computed by AGG. It is possible to observe that the potential conflicts are generated just to pairs of rules with the same priority. The reason for this is that we never can apply in the same graph rules with different priorities. This is because rule `bunch` and all rules of the second phase were not considered.

Although the tool has generated a large number of potential conflicts, in fact, none of the critical pairs represents a real conflict. This is a consequence of one of two possibilities: or the generated graphs are not reachable from the start state, since they do not respect one of the imposed restrictions to the initial graph; or the match of one of the rules does not satisfy one of its attribute conditions or one of its NACs. Figures



Minimal Conflicts

first \ second	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1 beginC		0	0																	
2 beginT	0		0																	
3 end	0	0																		
4 create					0	0	0	21	20	0	0	0	0							
5 createSeq				0		0	0	90	73	0	0	0	0							
6 createEnd				4	4		3	6	6	1	0	0	0							
7 count				0	0	3		7	6	0	0	0	0							
8 join				1	4	0	0		11	15	10	1	2							
9 joinCM				1	3	0	0	8		6	5	0	0							
10 broadcast				0	0	0	0	19	11		0	0	0							
11 broadcastSeq				1	6	0	0	14	10	10		2	9							
12 broadcastCM				0	0	0	0	0	0	0	0		0							
13 broadcastSeqCM				0	0	0	0	0	0	0	0	0								
14 spawn															10	28				
15 spawnSeq														10		18				
16 spawnAux														0	0					
17 conclude																		2		
18 concludeCM																	2			
19 endC																				0
20 endT																			0	

Figure 16. AGG Minimal Conflicts

17 to 19 exemplify different types of conflicting situations. Each figure illustrates a critical pair, where in the left-hand side is represented the graph G and in the right-hand side are detailed the conflicting rules. The matches of both rules in graph G is determined by vertices and edges numbering.

Figure 17 shows a potential conflict of type (i) between rules `conclude` and `concludeCM`: rule `conclude` deletes a tok edge which is in the match of `concludeCM`. This is a case where the generated graph G is not reachable from the start state, since it has two different tasks with the same id. In Figure 18 is found a potential conflict of type (ii) between `create` and `join`: rule `create` creates a seq_cre edge which is forbidden by NAC3 of rule `join`. However, the generated graph G does not respect the attribute conditions: task T of number 2 must be in a thread in order to rule `join` could be applied (attribute condition t2!=0 of `join`) and cannot be in a thread to rule `create` be applied (application condition thread=0 of `create`). In its turn, Figure 19 exhibits a potential conflict of type (iii) between rules `createEnd` and `create`: rule `createEnd` change the attribute value of vertex Count which is in the match of `create`. Nonetheless, the NAC1 of `createEnd`, which establishes that the task in evaluation could not have a successor task that is not in thread, is not satisfied by G.

In fact, all potential conflicts detected by AGG does not generate a graph G that is reachable and that satisfies all application conditions of both considered rules. Then, after the generation of all critical pairs by AGG and its individual analysis, we can

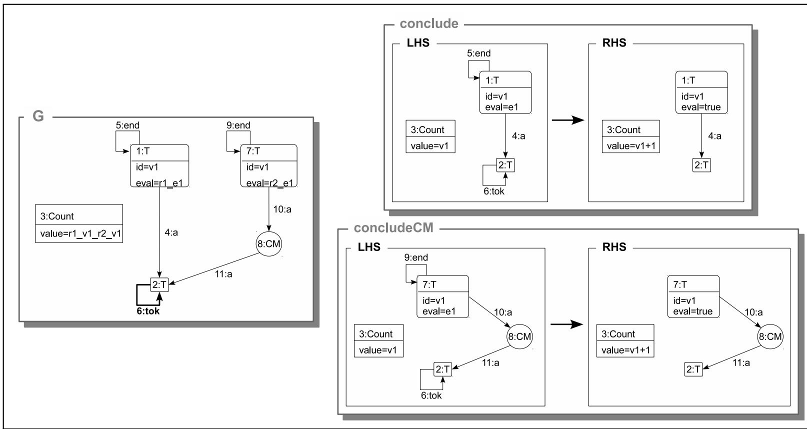


Figure 17. Potential Conflict Between conclude and concludeCM

assure the confluence of the graph grammar defined to transform DAGS in DCGs. We omit the individual analysis of each pair because it would be long and repetitive. Even if we consider an informal analysis, it is possible to deduce the confluence of the proposed transformation system. Follow a brief discussion by priorities groups.

- Rule `beginC` and `beginT` can never be applied to the same state graph (a NAC of `beginC` forbid the existence of a C task in the state graph and `beginT` requires it). Rule `end` just change an attribute that is not in the match of any other rule in the group. And rules `beginC` and `beginT` do not generate components that are in NAC of `end` nor delete elements, which would be in its match.
- Rules `create`, `join` and `joinCM` are specified to evaluate the task with lower id that was still not evaluated. Since they have mutually exclusive application conditions (for instance, rule `create` forbid that a successor task of the task in evaluation be in thread and rules `join` and `joinCM` require it), they will never be applied to the same state graph. Although `join` and `joinCM` have similar application conditions, both consider the successor task of the task in evaluation as the task of lower id, so just one of them can be matched. Rule `count` is just enabled when there is no task with id equals to the counter still not evaluated, then never conflicting with `create`, `join` or `joinCM`. Important to notice that when `create` is applied, rules `join`, `joinCM` and `count` will not be activated until the task in evaluation by `create` be set as evaluated and the counter incremented. This is because in order to a join be matched, the task in evaluation must have the id equals to the counter (and, as explained above, if a create was matched a join could not be matched). And to `count` be enabled, there should be

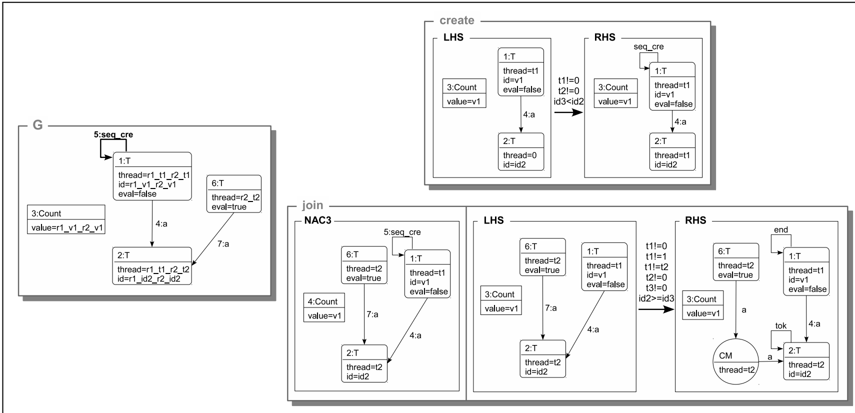


Figure 18. Potential Conflict Between create and join

no task with id equals to the counter still not evaluated (and this is the case if the task in evaluation was matched with create). In the same way, if a join is matched, rules `create` and `count` could not be matched until the task in evaluation be set as evaluated and the counter incremented. Remaining rules in the same group are just activated after the application of one of these rules. Rules `createSeq` and `createEnd` are just enabled after the execution of `create` and no other rule is activated together with them (because the others are just enabled after the execution of `join` or `joinCM`). They also have mutually exclusive application conditions (rule `createSeq` requires a successor task of the task in evaluation that not be in thread, and `createEnd` forbid it) and then could not be enabled simultaneously. Rule `broadcast` and `broadcastCM` are just active after a `join` application and rules `broadcastSeq` and `broadcastSeqCM` after `broadcast` or `broadcastCM` application. The reason is that `broadcast/ broadcastCM` requires a task with an end edge, created by `join` and `broadcastSeq/broadcastSeqCM` supposes the existence of a CM task created by `broadcast/broadcastCM`. Rules `broadcast` and `broadcastCM` do not delete any graph element and do not change any attribute. Besides, the elements created by each rule are not forbidden for the other. Rules `broadcastSeq` and `broadcastSeqCM` do not change any attribute and do not have any NAC. Both rules delete edges, but they have different types. So, the edge deleted by one of these rules never will be in the match of the other one.

- Rule `spawn` is just enabled after the execution of a `join` (its match supposes the existence of end and tok edges, created by `join`). Since it is in a lower priority

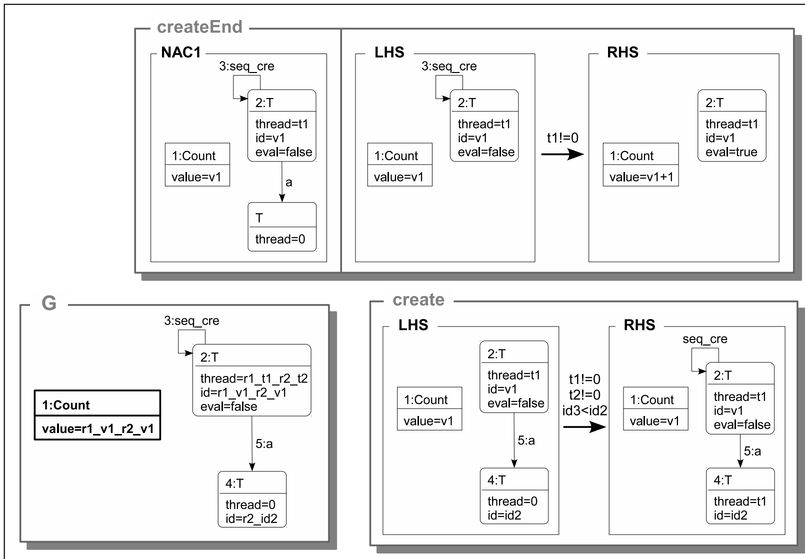


Figure 19. Potential Conflict Between createEnd and create

group it is applied after all broadcasts. Rules in this group also can never be active simultaneously. Rule `spawnSeq` and `spawnAux` require a CM task created by `spawn`. Rule `spawn` is applied just once, since it forbid the existence of a CM task created by its application. Rule `spawnAux` forbid that the successor of the task with end edge does not be in thread, while rule `spawnSeq` requires it.

- Rules `conclude` and `concludeCM` in the next group of priority are the rules that set the task in evaluation by a join as evaluated, incrementing the counter. Just one of them can be active, `conclude` in case of a `spawn` had not been applied to the task in evaluation and `concludeCM` in case of a `spawn` had been applied to the task in evaluation (`conclude` requires that the successor task of task with end edge be a T task and `concludeCM` requires that the successor task of task with end edge be a CM task).

- Rules `endC` and `endT` can never be applied to the same state graph, since a NAC of `endC` forbid the existence of a CF task in the state graph and `endT` requires it.

5 Case Studies

To exemplify the application of the transformation rules we are going to show along this section several examples using the transformation rules to obtain DCGs apart from DAGs found in literature.

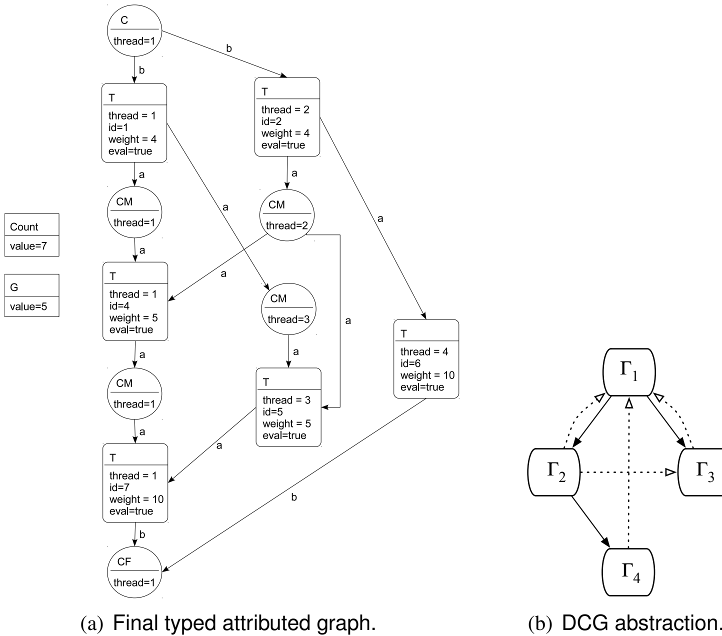
We applied the transformation rules to examples 1, 2, 3, 4, 5, 6, 7, 9, and 10 from Graham's work [4]. Here we named these graphs by $G_1, G_2, \ldots G_9$. Figure 20(a) shows the graph obtained after applying the first phase transformation rules to DAG G_2 depicted in Figure 1(a). G_2 is an interesting DAG for the transformation rules because the process uses the most grammar rules presented (except for *broadcast* rules – `broadcast`, `broadcastCM`, `BroadcastSeq`, and `BroadcastSeqCM`) to generate the final graph. Figure 20(a) presents the final typed attributed graph containing all the information about tasks and threads, whereas Figure 20(b) (obtained as the result of the second phase) shows a higher level abstraction of the same graph, the DCG, with thread information only. In this DCG, solid edges represent *create* operations, and dotted ones represent *join* operations. We can note that the DCG representation has less information (vertices and edges) than the original DAG, since threads encapsulate sequences of tasks. This is an important property for scheduling algorithms because a leaner graph represents fewer management in data structures involved in the scheduling, e.g. lists of *threads*.

Figure 21 presents C_i, i.e., the resulting graphs obtained by applying the first phase transformation rules for each graph G_i. The number inside each task represents its computational cost (weight). This cost can be presented in terms of the number m of available processors or in terms of ε, representing a very small cost; if no value is annotated, the cost is 1. Conforming tasks have no processing cost. Dashed rectangles are representing threads and dashed circles are describing conforming tasks. Note that graph C_2 is the same represented in Figure 20(a), except for some conforming tasks in thread 1. These conforming tasks were just added to simplify the transformation rules. Since they have only one incoming edge and one outcoming edge, just adding post-processing rules, they can be easily deleted.

5.1 Anahy's Scheduling Algorithm

Anahy's algorithm, which we are going to use to schedule the DGCs, manipulates five pools of threads: the **ready** pool, which stores references to threads that are able to run, and **running**, **finished**, **blocked**, and **unblocked** pools. The first three pools are global whereas the last two are local to the processors.

To better understand the scheduling algorithm, let's consider a machine with only one processor and an initial state where the DCG has only the vertex Γ_1, representing the main thread, and this thread is also in the ready pool. When the execution starts the processor is



(a) Final typed attributed graph. (b) DCG abstraction.

Figure 20. Resulting graph from DAG $G2$

idle, so the scheduler removes Γ_1 from the ready pool, schedules it over the processor, and inserts Γ_1 in the running pool. When a *create* is executed a new thread Γ_2 is inserted in the DCG and also in the ready pool. The processor continues Γ_1 until it executes a join over Γ_2. That is, at this point of Γ_1's execution, the processor needs Γ_2 to be completely executed so that it can resume Γ_1's execution. As Γ_2 is not in the finished pool, the processor inserts Γ_1 on its blocked pool (Γ_1 continues in the global ready pool) and asks the scheduler for work. Then the scheduler transfers Γ_2 from the ready pool to the running pool and schedules Γ_2 over the processor, which in turn starts Γ_2's execution. This scenario can repeat recursively, until Γ_2 is finished and the processor can safely take Γ_1 from the local blocked pool and resume its execution.

When we have multiple processors, they all execute simultaneously the algorithm described in the previous paragraph, so two or more threads can be running in parallel. So when a processor wants to execute a task in Γ_i that requires Γ_j to be finished that is, the processor reached a *join* scheduling point in thread Γ_i, two situations can occur: *(i)* Γ_j is on the finished pool, so the processor can read the data produced by Γ_j and resume the execution of Γ_i; *(ii)* Γ_j is not in the finished pool (it can be ready or already running on

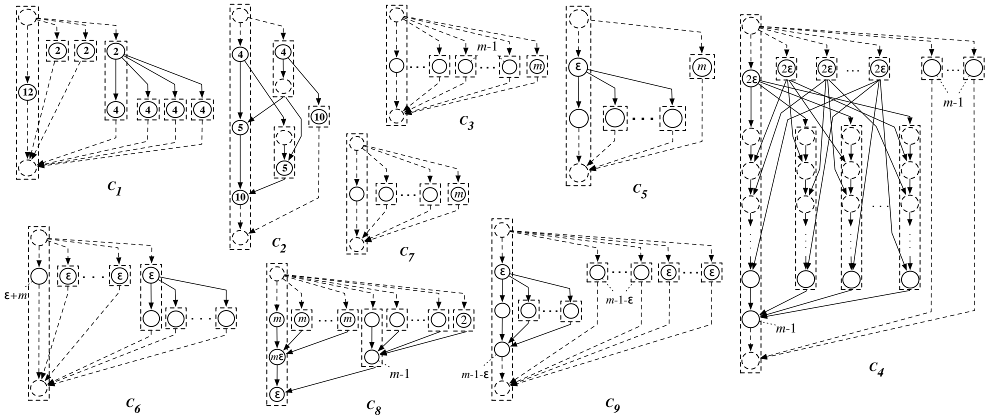


Figure 21. Resulting DCGs using Graham's DAGs as input for the grammar.

another processor), so the processor inserts Γ_i on its blocked pool and asks the scheduler for a job; Once this job is finished the processor can check if Γ_j is already finished and, eventually, transfer Γ_i to the unblocked pool and resume its execution.

Anahy's scheduler handles work requests differently when a processor gets idle because it has no work to do and when a processor gets idle because it has threads on its blocked pool and they can't be resumed. In the first case, the scheduler searches for a work in a breadth-first order from the root of the DCG, that is, the search prioritizes those threads that are closer to Γ_1, expecting the *critical path* to be as close to Γ_1 as possible. In the second case the scheduler searches the DCG the same way, but it considers the last blocked thread on the processor asking for a job as the root of the subgraph. If this search doesn't return results, the scheduler restarts the search from Γ_1, as in the first case, but ignoring the subgraph searched before. These two graph searches, however, consider only the edges inserted by *create* operations.

5.2 Schedule lengths

Using the transformation rules to obtain DCGs we can have a fair comparison among static DAG scheduling algorithms and multithread scheduling algorithms. So once we have the resulting DCGs we are going to schedule them using Anahy's [13] multithread scheduling algorithm.

The performance of this case study is presented in terms of the schedule length, i.e.,

Table 1. Scheduling results for G_i and C_i.

DCG/DAG	# m	$\|Anahy(C_i)\|$	$\|Opt(G_i)\|$	$\|S(G_i)\|$	$\|Anahy(C'_i)\|$
C_1/G_1	3	12	12	14	–
C_2/G_2	2	19	19	–	–
C_3/G_3	m	$2m-1$	m	$2m-1$	m
C_4/G_4	m	$2m-1+2\varepsilon$	$m+2\varepsilon$	–	$m+2\varepsilon$
C_5/G_5	m	$m+\varepsilon$	$m+\varepsilon$	–	–
C_6/G_6	m	$m'+2\varepsilon$	$m'+2\varepsilon$	–	–
C_7/G_7	m	$2m-1$	m	$2m-1$	m
C_8/G_8	m	$2m+\varepsilon$	$(m+1)(1+\varepsilon)$	$2m+\varepsilon$	–
C_9/G_9	m	$2m-1-2\varepsilon$	m	$2m-1-2\varepsilon$	–

the amount of time units required to execute the corresponding application, using a given scheduling algorithm. Table 1 presents the execution times for graphs from Figure 21 and the original ones. The first column indicates the corresponding graph (C_i/G_i) received as input by the Anahy and Graham scheduling algorithms, respectively. The second column shows the number m of processors considered for the schedule. The third column presents the execution time $|Anahy(C_i)|$ achieved by applying Anahy's scheduling strategy to the resulting DCGs. The fourth column presents the optimal schedule $|Opt(G_i)|$ for a given DAG whereas the fifth column $|S(G_i)|$ shows the schedule length obtained for the same DAG G_i changing the priority list in the algorithm. The last column presents execution times for Anahy scheduling algorithm after rearranging some callings of thread operations. The changes in the graphs and the algorithms will be later explained.

In order to demonstrate Anahy's algorithm efficiency scheduling the resulting DCGs, we compare our results with the best ones in Graham's work. Anahy's scheduling provides graphs C_1, C_2, C_5 and C_6 the same performance as the best possible performance achieved by scheduling the corresponding DAG. On the other hand, performance of our strategy is worse for the graphs C_3, C_4, C_7, C_8 and C_9.

The mechanism matches the patterns in a breadth-first order, so the generated DCG does not always take advantage of the scheduling technique used by Anahy's execution environment, as in G_3, G_4 and G_7. However, if the programmer knows the behavior of the algorithm that will schedule the DCG, (s)he can carefully describe the relationship between tasks to better explore the scheduling strategy. Based on this we can rewrite graphs C_3, C_4 and C_7 adding some precedence constraints to the original graph, and transform them into the graphs on Figure 22, with execution times $|Anahy(C'_i)|$ shown in Table 1.

For graphs C_8 and C_9, since the scheduler S (on the fifth column) implements

a critical path heuristic for these examples, we have $|Anahy(C_8)| = |S(G_8)|$ and $|Anahy(C_9)| = |S(G_9)|$. That means that Anahy's algorithm produces the same result as expected by a typical greedy list scheduling algorithm that takes into account the critical path.

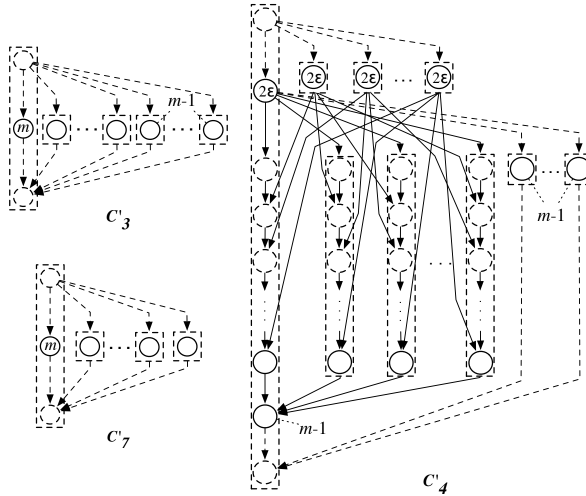


Figure 22. Result of transformation process over graphs G_3, G_4 e G_7 rewritten.

5.3 Grammar applications

The immediate use of the graph grammar is, by translating parallel programs' representations, to allow the analysis on several scheduling heuristics working on different program structures. Extending this analysis, if the programmer can also annotate the computational cost of each vertex, you can use the transformation mechanism proposed to convert a program written in a programming interface dataflow in a multithreaded program.

6 Conclusion

List scheduling algorithms are well-known strategies used to schedule parallel applications described in a DAG. Precursor works like Graham's [4] are the basis of many present works, even in scenarios where the program is not described as a DAG. In these scenarios, however, a DAG structure can be obtained from the interactions between the

programming interface, creating and destroying concurrent activities, and the scheduling environment.

This work presents an approach to transform a DAG into a DCG, which describes an equivalent program developed in a multithreaded fashion. To do so, we have designed a graph grammar that has been validated through practical tests, using DAGs found in literature and many other generated by an auxiliary tool (AKSSIM [27]). Such approach allows us to compare fairly scheduling algorithms that use the different graph representations (DAG or DCG). Case studies showed that multithread scheduling algorithms applied to DCGs obtained from our graph transformation process can be competitive with basic list strategies applied to the input DAGs. Furthermore a DCG is a leaner graph than a DAG describing the same program; hence multithread scheduling algorithms will have less complexity in comparison to list scheduling strategies applied to DAGs. The use of graph grammars also allowed the verification of some properties about the translation. Particularly, existence and uniqueness of the out coming model can be assured.

In future works, we intend to extend DCG's representation towards the description of the data sets transferred among threads. This will help us to specify and evaluate scheduling strategies that explore data locality, to schedule multithread applications on non-uniform memory access architectures. Moreover, we plan to analyze another kind of properties, which allow establishing relations between the source and the target graphs of the mapping.

Acknowledgement

The authors gratefully acknowledge financial support received from CNPq and FAPERGS (ARD-10/0348-8, ARD-11/0764-9, PRONEX-"Green Grid"). The author Cícero Augusto de S. Camargo is supported by grant CAPES (Coordenadoria de Aperfeiçoamento de Pessoal de Nível Superior).

References

[1] T. L. Casavant, Jon, and G. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Transactions on Software Engineering*, vol. 14, pp. 141–154, 1988.

[2] D. Feitelson, "Job scheduling in multiprogrammed parallel systems," *IBM Research Report*, vol. 19790, 1997.

[3] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Comput. Surv.*, vol. 36, no. 1, pp. 1–34, 2004.

[4] R. L. Graham, *Bounds on the Performance of Scheduling Algorithms*, ch. 5. John Wiley & Sons, 1976.

[5] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, vol. 31, pp. 406–471, Dec. 1999.

[6] M. JIS, "Computers and intractability a guide to the theory of np completeness," 1979.

[7] E. G. Coffman, *Computer and Job Shop Scheduling Theory*. New York: John Wiley & Sons Inc, 1976.

[8] T. Hu, "Parallel sequencing and assembly line problems," *Operations research*, vol. 9, no. 6, pp. 841–848, 1961.

[9] T. Yang and A. Gerasoulis, "Dsc: Scheduling parallel tasks on an unbounded number of processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 951–967, 1994.

[10] C. Intel, "Using intel(r) cilk(tm) plus." http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/cpp/mac/cref_cls/common/cilk_bk_using_cilk.htm, 2012. Acessado em Janeiro/2012.

[11] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP*. San Francisco: Morgan Kaufmann, 2001.

[12] J. Reinders, *Intel threading building blocks*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., first ed., 2007.

[13] G. G. H. Cavalheiro, L. P. Gaspary, M. A. Cardozo, and O. C. Cordeiro, "Anahy: A programming environment for cluster computing," in *VII High Performance Computing for Computational Science*, (Berlin), Springer-Verlag, 2007. (LNCS 4395).

[14] T. Gautier, X. Besseron, and L. Pigeon, "Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors," in *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pp. 15–23, ACM, 2007.

[15] L. G. Valiant, "A bridging model for parallel computation," *Communications of ACM*, vol. 33, no. 8, 1990.

[16] G. Rozenberg, ed., *Handbook of graph grammars and computing by graph transformation: volume I. foundations*. River Edge, USA: World Scientific Publishing Co., 1997.

[17] "AGG: The homebase.." http://user.cs.tu-berlin.de/ gragra/agg/. last access: November, 2011.

[18] C. Ermel, M. Rudolf, and G. Taentzer, *The AGG approach: language and environment*, pp. 551–603. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1999.

[19] C. A. S. Camargo, G. G. H. Cavalheiro, L. Foss, and S. A. C. Cavalheiro, "Uma gramática para a transformação de DAGs em grafos descrevendo programas multithreaded," in *WEIT 2011 - I Workshop-Escola de Informatica Teorica - Anais*, pp. 164–176, 2011.

[20] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, *Fundamentals of Algebraic Graph Transformation (An EATCS Series).* NJ, USA: Springer-Verlag, Inc., 2006.

[21] H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, eds., *Handbook of graph grammars and computing by graph transformation: volume III. Concurrency, parallelism, and distribution*. River Edge, USA: World Scientific Publishing Co., 1999.

[22] D. B. Shmoys, J. Wein, and D. P. Williamson, "Scheduling parallel machines on-line," *SIAM J. Comput.*, vol. 24, no. 6, pp. 1313–1331, 1995.

[23] R. Fleischer and M. Wahl, "On-line scheduling revisited," *Journal of Scheduling*, p. 343–353, 2000.

[24] S. Albers, "Better bounds for on-line scheduling," *SIAM Journal on Computing*, p. 459–473, 1999.

[25] H. El-Rewini and T. G. Lewis, "Scheduling parallel program tasks onto arbitrary target machines," *J. Parallel Distrib. Comput.*, vol. 9, no. 2, pp. 138–153, 1990.

[26] R. D. Blumofe and et al., "Cilk: An efficient multithreaded runtime system," *ACM SIGPLAN Not.*, vol. 30, Aug. 1995.

[27] C. A. S. Camargo, A. S. Araújo, and C. G. G. H., "Akssim: Uma ferramenta para a análise de algoritmos de lista em ambientes multithreaded dinâmicos," in *ERAD 2011 - XI Escola Regional de Alto Desempenho*, (Porto Alegre), SBC, 2011.