Scheduling Dynamic Graphs*

Andreas Jakoby[1,3] Maciej Liśkiewicz[2] Rüdiger Reischuk[1]
[1]Med. Universität zu Lübeck [2]Universität Tübingen

April 1998

Abstract

In parallel and distributed computing scheduling low level tasks on the available hardware is a fundamental problem. Traditionally, one has assumed that the set of tasks to be executed is known beforehand. Then the scheduling constraints are given by a *precedence graph*. Nodes represent the elementary tasks and edges the dependencies among tasks. This static approach is not appropriate in situations where the set of tasks is not known exactly in advance, for example, when different options how to continue a program may be granted.

In this paper a new model for parallel and distributed programs, the *dynamic process graph,* will be introduced, which represents all possible executions of a program in a compact way. The size of this representation is small – in many cases only logarithmically with respect to the size of any execution. An important feature of our model is that the encoded executions are directed acyclic graphs having a "regular" structure that is typical of parallel programs. Dynamic process graphs embed constructors for parallel programs, synchronization mechanisms as well as conditional branches. With respect to such a compact representation we investigate the complexity of different aspects of the scheduling problem: the question whether a legal schedule exists at all and how to find an optimal schedule. Our analysis takes into account communication delays between processors exchanging data.

Precise characterization of the computational complexity of various variants of this compact scheduling problem will be given in this paper. The results range from easy, that is $\mathcal{NLOGSPACE}$-complete, to very hard, namely $\mathcal{NEXPTIME}$-complete.

Keywords: ALT-PAR-program constructors, precedence graph, succint graph representation, scheduling, communication delay

1 Introduction

Scheduling tasks efficiently is crucial for fast executions of parallel and distributed programs. An intensive study of this scheduling problem has led to the development of a number of algorithms that cover a wide spectrum of strategies: from fully static, where the compiler completely precomputes the schedule, i.e. when and where each task will be executed, to fully dynamic, where tasks are scheduled at run-time only. Changing from static to dynamic strategies one gets the potential of reducing the total execution time of a program because the resources are better used, but in general there will be more effort necessary at run-time. Therefore, existing parallel systems fix most details of the schedule already at compile time (see e.g. [7]). For a more dynamic and also fault tolerant approach see for example the MAFT project [12, 11].

In many cases, the set of tasks that have to be executed is not precisely known at compile time. El-Rewini and Ali have introduced a parallel program model that allows a suitable data representation for static scheduling algorithms [3]. The representation is based on two directed graphs: the *branch graph* and the *precedence graph.* This approach models conditional branchings quite well, but it is unsuited for parallel program constructors or synchronization mechanisms, for example the channel concept as implemented in the parallel programming language OCCAM.

*supported by DFG Research Grant Re 672/2
[1]Institut für Theoretische Informatik, email: jakoby/reischuk@informatik.mu-luebeck.de
[2]Wilhelm-Schickard Institut für Informatik, email liskiewi@informatik.uni-tuebingen.de,
on leave from Instytut Informatyki, Uniwersytet Wrocławski
[3]**contact author:** A. Jakoby, Institut für Theoretische Informatik, Wallstr. 40, D-23560 Lübeck, Germany,
tel.: +49 (451) 7030413, fax.: +49 (451) 7030438

1.1 Extension to a Dynamic Environment

In this paper we introduce a new model, the *dynamic process graph,* **DPG**, which allows a natural representation for parallel and distributed programs. In particular, it gives a concise description of static scheduling problems for highly concurrent programs. An important feature of DPGs is that they resemble the characteristics of executing typical parallel or distributed programs, and this in a space-efficient way. This representation can provide an exponential compaction compared to the length of execution sequences.

Our main technical contribution is to analyze the complexity of finding optimal schedules with respect to this compact program representation. We will concentrate on scheduling elementary tasks where each task has unit execution time. No bound will be put on the number of processors available.

1.2 Communication Delays

Papadimitriou and Yannakakis have argued in [15] that scheduling policies should take into account communication delays occurring when one processor sends a piece of data to another one. Thus, it will be faster to schedule dependent tasks on the same processor. For further results concerning scheduling with communication delays in the standard static setting see [10, 9, 16]. Here, we will extend the complexity analysis to the dynamic case.

Communication delays will be specified by a function $\delta : E \to \mathbb{N}$, which defines the time elapsed when the corresponding data is sent from one processor to another one. For simplification we assume that this delay is independent of the particular pair of processors (alternatively that $\delta(e)$ gives an upper bound on the maximal delay). Scheduling with communication delays requires the following condition to be fulfilled:

- if a task v is executed on processor p at time t then for each direct predecessor u of v holds: u has been finished either on p by time $t-1$, or on some other processor p' by time $t-1-\delta(u,v)$.

1.3 Dynamic Dependencies

In many systems, dependencies among particular task-instances are determined by a scheduling policy, not by the program itself. Consider the following situation. A program contains a part P where several processes concurrently generate data. The program can continue as soon as one of these results is available. Such a situation can compactly be described using the ALT-constructor of OCCAM [1]. In the piece of code given in Fig. 1 one process sends data down channel C1, while a second one behaves similarly using channel C2.

```
P
ALT
  C1 ? X
     P1
  C2 ? X
     P2
```

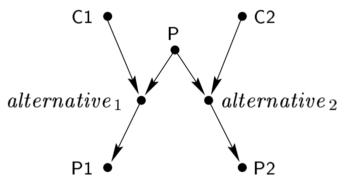

Figure 1: *A dynamic precedence graph representing* P*: the output mode of* P *is* ALT *and the input mode of alternative*$_1$ *and alternative*$_2$ *is* PAR.

A scheduling policy has to select one (and only one) of the alternatives Pi. Hence, either P1 or P2 will be a successor of P.

If both channels Ci get ready at the same time then a scheduler can choose arbitrarily. However, to minimize the total execution time it is helpful to know which process can be executed faster, P1 or P2? Even if the two channels do not get ready simultaneously executing a ready alternative immediately overall may lead to a longer schedule than waiting until the other channel is ready.

We will consider different degrees of concurrency expressed by the ALT constructor and the PAR constructor to create parallel processes, and analyze the complexity of the corresponding scheduling problems with respect to the amount of concurrency.

To represent such parallel programs in a natural way, we introduce dynamic process graphs, which are generalizations of standard precedence graphs. A dynamic process graph is an acyclic graph $G=(V,E)$ with two sets of labels $I(v), O(v) \in \{\mathsf{PAR}, \mathsf{ALT}\}$ attached to the nodes $v \in V$. Nodes represent tasks, edges dependencies between tasks. A complete formal definition will be given in the next section.

The label $I(v)$ describes the *input mode* of task v. If $I(v) = \mathsf{ALT}$ then to execute v at least one of the predecessor tasks u with $(u,v) \in E$ has to be completed. $I(v) = \mathsf{PAR}$ requires that executions of all predecessors of v have to be completed before v can start. If task v has been completed then according to the *output mode* $O(v)$ one of v's successors in case $O(v) = \mathsf{ALT}$ (resp. all of them in case $O(v) = \mathsf{PAR}$) has to be initiated. Fig. 1 gives an example of such a representation.

Dynamic process graphs are a compact way to illustrate data dependencies of parallel programs. In particular, it will be easy for programs written in a parallel programming language like OCCAM or Ada. Note that a standard precedence graph cannot represent such programs in a simple way. We should note that dynamic process graphs can also be modeled by a certain class of Petri nets and their reachability problem. However, this class does not seem to correspond to those subclasses that have been considered in more detail in the literature. We will therefore stick to the DPG model.

1.4 New results

The scheduling problem for dynamic process graphs is a natural generalization of the static scheduling problem. In particular, the delay scheduling for a precedence graph G is equivalent to the scheduling problem for the dynamic process graph (G, I, O) with $I, O \equiv \mathsf{PAR}$. In the static case, the scheduling problem with communication delays is already computationally difficult. In [15] it has been shown that this problem is $\mathcal{NP}$-complete even if for each graph the communication delay takes only a single value, but this value has to increase with the size of the graph. We have improved this result in [9] showing that the problem remains $\mathcal{NP}$-complete even if we restrict the class of precedence graphs to (1,2)-trees. On the other hand, in [10] it has been shown that for fixed delay $\delta \equiv c$ independent of the precedence graphs the problem can be solved in polynomial time where the degree of the polynomial grows with c.

Due to the compact representation in our dynamic model it is no longer obvious that the dynamic scheduling problem can be solved in $\mathcal{NP}$ at all. In fact, one of our main results is that even restricted to constant communication delay the scheduling problem for dynamic process graphs is $\mathcal{NEXPTIME}$-complete. To prove this we construct a reduction of the SUCCINCT-3SAT problem. However, if we restrict the input mode I to ALT then the problem becomes $\mathcal{P}$-complete. Even more, also fixing the output mode the problem becomes $\mathcal{NLOGSPACE}$-complete. A similar complexity jump has been observed for classical graph problems in [4, 14, 13]. They show that simple graph properties become $\mathcal{NP}$-complete when the graph is represented in a particular succinct way using generating circuits or a hierarchical decomposition. Under the same representation graph properties that are ordinarily $\mathcal{NP}$-complete, like HAMILTON CYCLE, 3-COLORABILITY, CLIQUE (of size $|V|/2$), etc., become $\mathcal{NEXPTIME}$-complete.

On the other hand, some restricted variants of this scheduling problem, which may seem to be easy at first glance, remain hard, namely $\mathcal{NP}$-complete. Fig. 2 summarizes our results about the complexity of the dynamic delay scheduling problem with respect to the input and output modes that may occur in the graphs.

input mode	output mode	complexity
ALT	ALT	$\mathcal{NLOGSPACE}$-complete
ALT	PAR	$\mathcal{NLOGSPACE}$-complete
ALT	$\mathsf{ALT}, \mathsf{PAR}$	$\mathcal{P}$-complete
PAR	$\mathsf{ALT}, \mathsf{PAR}$ or PAR or ALT	$\mathcal{NP}$-complete
$\mathsf{ALT}, \mathsf{PAR}$	ALT	$\mathcal{NP}$-complete
$\mathsf{ALT}, \mathsf{PAR}$	PAR	$\mathcal{BH}_2$-hard
$\mathsf{ALT}, \mathsf{PAR}$	$\mathsf{ALT}, \mathsf{PAR}$	$\mathcal{NEXPTIME}$-complete

Figure 2: *The complexity of dynamic scheduling with communication delays with respect to input and output modes.*

We will also consider the question whether for a given dynamic process graph there exists a run for a program represended by the graph at all. It will be shown that the problem is $\mathcal{NP}$-complete even if the input mode can only be PAR and the output mode only ALT.

The remaining part of this paper is organized as follows. In Section 2 we give some examples and a formal definition of DPGs and their scheduling problem. Section 3 studies the complexity of the existence problem. The last section deals with the problem to find optimal schedules. Due to space limitations we will only present the technically most difficult result, which is scheduling unrestricted DPGs, and give a short sketch of the construction for the exponential lower bound.

2 Dynamic Process Graphs and Runs

For illustration, consider the following parallel program P written in OCCAM (Fig. 3). This program contains branches, but the situation is not as bad as it could be since the branching does not depend on the current values of variables. Still there is the problem to determine the set of tasks that have to be executed at run time. It will turn out that even for such restricted programs the scheduling problem is quite hard. Depending on the ALT branches chosen the execution of this program is represented by one of the four possible runs shown in Fig. 4. The following definition tries to capture this dichotomy between parallel/distributed programs and their executions.

```
SEQ
   ALT
      in1 ?  x
         P1
      in2 ?  x
         P2
   PAR
      P3
      P4
   ALT
      in1 ?  y
         P5
      in2 ?  y
         P6
```

Figure 3: *OCCAM program* P

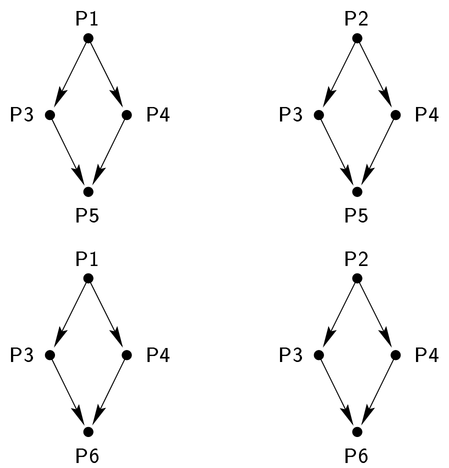

Figure 4: *Possible runs of* P.

Definition 1 *A* **dynamic process graph** $\mathcal{G} = (G, I, O)$ *is a directed acyclic graph (DAG)* $G = (V, E)$, *with node labellings* $I, O : V \to \{\mathsf{ALT}, \mathsf{PAR}\}$. $V = \{v_1, v_2, \ldots, v_n\}$ *represents a set of* **processes** *and* E *dependencies among them.* I *and* O *describe* **input modes**, *(resp.* **output modes***) of the processes.*

A finite DAG $H_{\mathcal{G}} = (W, F)$ *is a* **run** *of* $\mathcal{G}$ *iff the following conditions are fulfilled:*

1. *The set* W *is partitioned into subsets* $W(v_1) \cup W(v_2) \cup \ldots \cup W(v_n)$. *The nodes in* $W(v_i)$ *are* **execution instances** *of the process* v_i *and will be called* **tasks**.

2. *Each source node of* G *has exactly one execution instance in* $H_{\mathcal{G}}$.

3. *For a process* $v \in V$ *let pred*$(v) := \{u_1, u_2, \ldots, u_p\}$ *denote the set of all predecessors of* v *and succ*$(v) := \{w_1, w_2, \ldots, w_r\}$ *its successors. For any execution instance* x *of* v *in* $W(v)$ *it has to hold*
   - *if* $I(v) = \mathsf{ALT}$ *then* x *has a unique predecessor* y *belonging to* $W(u_i)$ *for some* $i \in \{1, \ldots, p\}$;
   - *if* $I(v) = \mathsf{PAR}$ *then pred*$(x) = \{y_1, y_2, \ldots, y_p\}$ *with* $y_i \in W(u_i)$ *for each* i.
   - *if* $O(v) = \mathsf{ALT}$ *then* x *has a unique successor* z *belonging to* $W(w_j)$ *for some* $i \in \{1, \ldots, r\}$;
   - *if* $O(v) = \mathsf{PAR}$ *then succ*$(x) = \{z_1, z_2, \ldots, z_r\}$ *with* $z_j \in W(w_j)$ *for each* j.

For a DPG $\mathcal{G}$ *with run* $H_{\mathcal{G}} = (W, F)$ *define* $W(v, v') := \{(y, z) \in F \mid y \in W(v)$ *and* $z \in W(v')\}$.

Fig. 5 shows a node v of a DPG with input mode Q_1 and output mode Q_2. Through the paper we will illustrate the ALT-mode by a white box, the PAR-mode by a black box. For a source or a node with indegree 1 the input mode is obviously inessential. Hence we will ignore such a label, and similarly the output label in case of a sink or a node with outdegree 1. A dynamic process graph corresponding to the program of Fig. 3 is shown in Fig. 6 (a).

Observe that a run can be smaller than its defining dynamic process graph, e.g. the graph in Fig. 6 (a) has 10 nodes while its run (b) only 8 since certain tasks are not executed at all. More typically, however, a run will be larger than the dynamic process graph itself since the PAR-constructor allows process duplications. If, for example, the output mode of vertex v_1 in Fig. 6 is changed from ALT into PAR then both processes P_1 and P_2 have to be executed. Hence, process v_2 with input mode ALT has to be duplicated in order to consume both processes. This

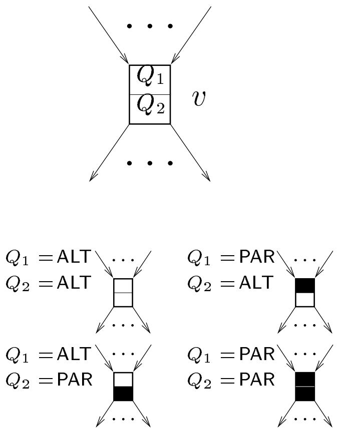

Figure 5: *A node v with input label Q_1 and output label Q_2 and the schematic representation.*

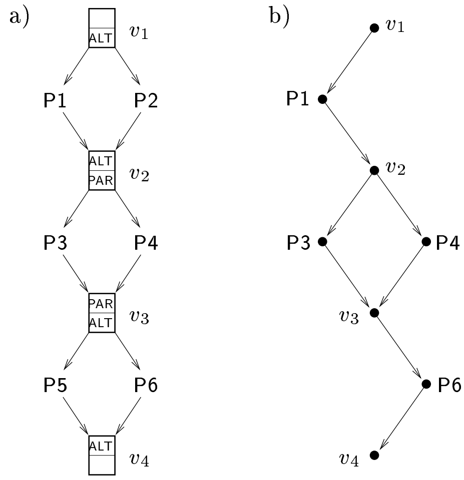

Figure 6: *(a) a dynamic process graph for program* P*, (b) a run of this dynamic process graph.*

in turn implies that P_3 and P_4 will have 2 execution instances both. Therefore, each run of the modified graph consists of 15 nodes. The following lemma gives an upper bound on this blow-up, resp. the possible compaction ratio of dynamic process graphs.

Lemma 1 *Let $\mathcal{G} = ((V,E),I,O)$ be a dynamic process graph and $H_{\mathcal{G}} = (W,F)$ be a corresponding run. Then it holds $|W| \le 2^{|V|-1}$. Moreover, this general bound is tight.*

Proof. Call a node v of $G = (V,E)$ to be in level ℓ if the length of the longest path between a source and v is $\ell - 1$, and let r_ℓ, denote the number of nodes on level ℓ. Let $H_{\mathcal{G}}$ be any run of $\mathcal{G}$ and let $R(\ell)$ denote the total number of execution instances in $H_{\mathcal{G}}$ for processes $v \in G$ in levels with index at most ℓ. We claim that

$$R(\ell) \le (r_\ell + 1)(r_{\ell-1} + 1) \ldots (r_2 + 1) r_1 \ . \tag{1}$$

For $\ell = 1$ it holds $R(1) = r_1$ since according to the definition of arun each source of G has exactly one execution instance in $H_{\mathcal{G}}$. Moreover, if we assume that the inequality (1) holds for $\ell < i$ then $R(i)$ can be bounded by

$$R(i) \ \le \ R(i-1) \ + \sum_{v \text{ of level } i} \ \sum_{u \in in(v)} |W(u)| \ \le \ R(i-1) + r_i \cdot R(i-1) \ ,$$

since each node v in level i has its predecessors in levels $i-1, \ldots, 2, 1$ (in the worst case, all nodes from this levels can be predecessors of v). Hence, the total number of execution instances of v does not exceed $R(i-1)$.

It is easy to check that among all sequences of positive integers $r_1, r_2, \ldots, r_\ell$, fulfiling the condition $r_1 + r_2 + \ldots + r_\ell = |V|$ the value $(r_\ell + 1)(r_{\ell-1} + 1) \ldots (r_2 + 1) r_1$ from the inequality (1) is maximal when all numbers r_i are equal to 1, which gives $|W| \le 2^{|V|-1}$.

To see that the bound is tight consider the complete DAG on n nodes $v_1, \ldots, v_n$ with edges (v_i, v_j) for all $1 \le i < j \le n$. All input labels are ALT, while all output labels are PAR. It is easy to see that this dynamic process graph has only one run, and this has size 2^{n-1}. ∎

Lemma 2 *Any dynamic process graph has at most double exponential many different runs and this bound can actually occur.*

Definition 2 *Let $\mathcal{G} = (G,I,O)$ be a dynamic process graph with communication delay $\delta : E \to I\!N$ between its processes. A schedule for $\mathcal{G},\delta$ is a schedule of a run $H = (W,F)$, where the communication delay between each execution instance $x \in W(u)$ and $y \in W(v)$ is given by $\delta(u,v)$.*

If S is a schedule of $\mathcal{G},\delta$ then let $T(S)$ denote the duration of S, i.e. the amount of time necessary to complete all tasks in S. Define $\boldsymbol{T_{opt}(\mathcal{G},\delta)} \ := \ \min_{S \text{ for } \mathcal{G},\delta} T(S)$.

This leads to the following decision problem:

Definition 3 DYNAMIC PROCESS GRAPH SCHEDULE (DPGS)
Given a DPG (G,I,O), a communication delay δ, and a deadline T^*, does $T_{opt}(\mathcal{G},\delta) \ \le \ T^*$ hold?

3 The Execution Problem

The number of different runs of a DPG can be huge according to Lemma 2. On the other hand, it is not obvious that for any DPG an appropriate run exists at all. It is easy to see that dynamic process graphs with either only PAR labels, or with all input labels equal to ALT can always be executed. The first case corresponds to standard static precedence graphs. However, this is no longer true for arbitrary DPGs. For a simple example of a graph which has no run see Fig. 7. In this case the input mode of all nodes is PAR and their output mode ALT. This section studies the problem whether for a given dynamic process graph there exists a run or not. Obviously, a schedule for a given DPG can be constructed if an appropriate run exists.

Definition 4 Execution Problem for DPGs (ExDPG)
Given a DPG $\mathcal{G}$, decide whether it can be executed, that means whether it has run.
ExDPG[IN | OUT] *denotes the execution problem for the class of DPGs with input and output modes restricted to* IN *(resp.* OUT*), where* IN, OUT $\subseteq \{\mathsf{ALT}, \mathsf{PAR}\}$.

As we have seen above for some restricted DPGs this question has a trivial answer. In general, a decision procedure may be complex. Of course, if a DPG has a run then it has also a schedule, and since we have estimated a bound on the size of the run, one can also compute an upper bound on the maximal schedule length given the maximal communication delay. Thus, the execution problem for DPGs could be solved by a reduction to the scheduling problem with a huge enough deadline. However, we want to capture the complexity of the execution problem more precisely and thus will investigate it directly.

The main negative result of this section says that for graphs with arbitrary input and output modes the execution problem is $\mathcal{NP}$-complete. We will prove even more, namely the problem remains $\mathcal{NP}$-complete even if the input mode of all nodes is PAR while the output one is ALT. On the positive side, the complexity decreases drastically for DPGs with output labels restricted to PAR.

Definition 5 *Assume that $H_\mathcal{G} = (W, F)$ is a run of a DPG $\mathcal{G}$ with processes $V = \{v_1, \ldots, v_n\}$. The sequence $\mathcal{C} := (|W(v_1)|, |W(v_2)|, \ldots, |W(v_n)|)$ will be called the* **characteristic vector** *of $H_\mathcal{G}$.*

Lemma 3 *If $\mathcal{G}$ is a DPG with output mode restricted to* PAR *then all runs have exactly the same characteristic vector.*

Proof: Assume that $\mathcal{G}$ can be executed, that meanss it has at least one run. The output mode restriction implies that each process of $\mathcal{G}$ has to be executed because it is either a source or a successor of a process with output mode PAR. The proof uses induction on the depth of $\mathcal{G}$. Assume that the claim holds for the subgraph $\mathcal{G}'$ of $\mathcal{G}$ obtained by removing all nodes of maximal depth.

By definition, for all runs $H_\mathcal{G} = (W, F)$ and for all nodes v of maximal depth in $\mathcal{G}$ it holds: if u is a predecessor of v and $t_u \in W(u)$ an execution instance of u then there exists a unique task $t(t_u, v) \in W(v)$ such that $(t_u, t(t_u, v)) \in F$. If $I(v) = \mathsf{ALT}$ then the in-degree of $t(t_u, v)$ is 1, thus there are as many execution instances of v required as there are predecessor tasks t_u. If, by inductive assumption, the size of the $W(u)$ are independent of the particular run the same holds for the size of $W(v)$.

On the other hand, if $I(v) = \mathsf{PAR}$ then the in-degree of each execution instance of v has to equal the number of predecessors of v. In this case, there can only be a run for $\mathcal{G}$ if all predecessors u of v have the same number of execution instances. ∎

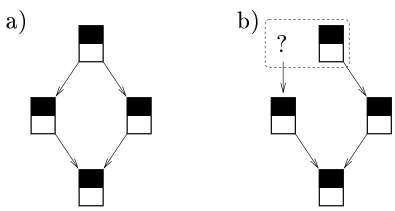

Figure 7: *Example of a dynamic process graphs with $O \equiv \mathsf{ALT}$ and $I \equiv \mathsf{PAR}$ that has no strategy.*

input mode	output mode	complexity
ALT	arbitrary	trivial
PAR	PAR	trivial
ALT, PAR	PAR	$\mathcal{NC}^2$
PAR	ALT	$\mathcal{NP}$-complete
ALT, PAR	ALT	$\mathcal{NP}$-complete
ALT, PAR	ALT, PAR	$\mathcal{NP}$-complete

Figure 8:

Theorem 1 *ExDPG*[ALT, PAR | PAR] $\in \mathcal{P}$.

The procedure below computes the characteristic vector of a DPG $\mathcal{G}$ if it can be executed; otherwise it returns `false`. Hence, it solves ExDPG[PAR,ALT | PAR]. Its time complexity is $O(|V|+|E|)$ assuming uniform time measure.

`procedure` *characteristic-vector* $(\mathcal{G})$
`compute a topological ordering` $(v_1, v_2, \ldots, v_n)$ `of` $\mathcal{G}$
`for` $i := 1$ `to` n `do`
  `if` v_i `is a source then` $A(v_i) := 1$
  `elseif` $I(v_i) = \mathsf{ALT}$ `then` $A(v_i) := \sum_{v_j \in \mathrm{pred}(v_i)} A(v_j)$
  `elseif` $A(u) = A(w)$ `for every` $u, w \in \mathrm{pred}(v_i)$ `then` $A(v_i) := A(u)$
  `else return false and halt`
`return` A

Using dynamic programming we can show that $ExDPG[\mathsf{ALT}, \mathsf{PAR} \mid \mathsf{PAR}]$ can be solved in $\mathcal{NC}^2$, which improves the bound above. Since the $\mathcal{NC}^2$ algorithm is technically more involved we cannot present details here and refer to the complete technical report [8].

Theorem 2 $ExDPG[\mathsf{ALT}, \mathsf{PAR} \mid \mathsf{PAR}] \in \mathcal{NC}^2$.

By an reduction of 3DM to $ExDPG[\mathsf{PAR} \mid \mathsf{ALT}]$ it can be shown

Theorem 3 $ExDPG[\mathsf{PAR} \mid \mathsf{ALT}]$ *is $\mathcal{NP}$-hard, even if the underlying DPG is planar and bipartite.*

The hardness of $ExDPG[\mathsf{ALT}, \mathsf{PAR} \mid \mathsf{ALT}]$ and $ExDPG[\mathsf{ALT}, \mathsf{PAR} \mid \mathsf{ALT}, \mathsf{PAR}]$ follows directly by inclusion.

Because of the double exponential bound on the number of different runs it is not obvious that the execution problem for unrestricted DPGs can be solved in $\mathcal{NP}$ at all. By Lemma 1, we know that the expansion between a DPG and its runs is at most exponential. Considering the characteristic vector of a run and the number of edges between execution instances of different processes a legal run can be characterized as follows:

Lemma 4 *iven a DPG $\mathcal{G}$ and values $A(v_i), A(v_i, v_j) \in \mathbb{N}$ for all nodes v_i and edges (v_i, v_j) of $\mathcal{G}$. Then there exists a run $H_{\mathcal{G}}$ for $\mathcal{G}$ with $|W(v_i)| = A(v_i)$ and $|W(v_i, v_j)| = A(v_i, v_j)$ iff*

1. $A(v_i) = 1$ *for all sources* v_i,
2. *for all non-sources v_i with* $I(v_i) = \mathsf{ALT}$: $A(v_i) = \sum_{(v_j, v_i) \in E} A(v_j, v_i)$,
3. *for all non-sources v_i with* $I(v_i) = \mathsf{PAR}$: $\forall (v_j, v_i) \in E \ : \ A(v_i) = A(v_j, v_i)$,
4. *for all non-sinks v_i with* $O(v_i) = \mathsf{ALT}$: $A(v_i) = \sum_{(v_i, v_j) \in E} A(v_i, v_j)$,
5. *for all non-sinks v_i with* $O(v_i) = \mathsf{PAR}$: $\forall (v_i, v_j) \in E \ : \ A(v_i) = A(v_i, v_j)$.

Proof: The *only if*-case follows directly from the definition of the sets $W(v_i)$ and $W(v_i, v_j)$.

For the *if*-case we have to notice that we can easily construct the sets $W(v_i)$ and $W(v_i, v_j)$ which fulfill the properties of a run for given values $A(v_i)$ and $A(v_i, v_j)$. ∎

This lemma provides the basis for the following nondeterministic procedure that solves the most general problem $ExDPG[\mathsf{ALT}, \mathsf{PAR} \mid \mathsf{ALT}, \mathsf{PAR}]$ in polynomial time.

`procedure` *solve-ExDPG*$(\mathcal{G})$
1 `for all sources` v_i `of` $\mathcal{G}$ `set` $A(v_i) := 1$
2 `for all non-sources` v_i `of` $\mathcal{G}$ `choose nondeterministically` $A(v_i) \in \{0, \ldots, 2^{|V|-1}\}$
3 `for all edges` (v_i, v_j) `of` $\mathcal{G}$ `choose nondeterministically` $A(v_i, v_j) \in \{0, \ldots, 2^{|V|-1}\}$
4 `for all non-sinks` v_i `of` $\mathcal{G}$
5   `if` $O(v_i) = \mathsf{ALT}$ `then verify that` $A(v_i) = \sum_{(v_i, v_j) \in E} A(v_i, v_j)$ `and reject if not`
6   `else verify that` $A(v_i) = A(v_i, v_j)$ `for all edges` $(v_i, v_j) \in E$ `and reject if not`
7 `for all non-sources` v_i `of` $\mathcal{G}$
8   `if` $I(v_i) = \mathsf{ALT}$ `then verify that` $A(v_i) = \sum_{(v_j, v_i) \in E} A(v_i, v_j)$ `and reject if not`
9   `else verify that` $A(v_i) = A(v_j, v_i)$ `for all edges` $(v_j, v_i) \in E$ `and reject if not`
10 `accept`

For the correctnes of this procedure note that a characteristic vector A of a run $H_{\mathcal{G}}$ will be guessed in lines 1 and 2 if $\mathcal{G}$ can be executed. To check that this is a valid one the auxiliary values $A(u, v)$ are guessed in line 3 for all edges (u, v) of $\mathcal{G}$. It is easy to see that the rest of the procedure verifies the valildity of this characteristic vector. On the other hand, if $\mathcal{G}$ has no run then for any guess of values $A(v_i)$ some inconsistency will be detected. Obviously, *solve-ExDPG* works in polynomial time.

Theorem 4 $ExDPG[\mathsf{ALT},\mathsf{PAR} \mid \mathsf{ALT},\mathsf{PAR}] \in \mathcal{NP}$.

One can show that $ExDPG[\mathsf{ALT},\mathsf{PAR} \mid \mathsf{ALT}]$ remains $\mathcal{NP}$-complete even for DPGs with only one sink. On the other hand, these problems become easy if we consider DPGs with only one source. In this case, ExDPG with $O \equiv \mathsf{ALT}$ is equivalent to the decision problem whether there exists a path from the source to a sink where all nodes on the path with in-degree at least two have input mode ALT. Hence, a solution of $ExDPG[\mathsf{PAR} \mid \mathsf{ALT}]$ consists of chains only and is therefore solvable by a finite automata. By a reduction of the *DAG Graph Accessibility Problem [DAG-GAP]* we can show that the single source $ExDPG[\mathsf{ALT},\mathsf{PAR} \mid \mathsf{ALT}]$-problem is $\mathcal{NLOGSPACE}$-complete.

4 Scheduling Dynamic Process Graphs

Let us now consider the problem to construct optimal schedules for dynamic process graphs. Since the execution problem is already hard in the less restricted cases one has to expect similar negative results for the scheduling problem. Our main result below, however, implies that the compaction provided by dynamic process graphs is quite efficient. The complexity of scheduling general DPGs increases to $\mathcal{NEXPTIME}$-complete. The hardness proof will be the topic of this section. We have also anlaysed the complexity for restricted classes of DPGs. They will be mentioned at the end of this section.

Theorem 5 *Scheduling dynamic process graphs is $\mathcal{NEXPTIME}$-hard, even in case of constant communication delay.*

Proof. For the reduction we will use the following problem

Definition 6 SUCCINCT-3SAT: *As input we are given a Boolean circuit over the standard AND ,OR ,NOT - basis that succinctly codes a Boolean formula in conjunctive normal form with the additional property that each clause has exactly three literals and each literal appears exactly three times. Suppose that the encoded formula consists of n variables and m clauses. On input $(0,i,k)$ with $i \in \{0,\ldots,n-1\}$ and $k \in \{1,2,3\}$ (appropriately coded in binary), the coding circuit returns the index of the clause where the literal $\neg x_i$ appears the k-th time. On input $(1,i,k)$ it returns the index of the clause where x_i appears for the k-th time. On input $(2,j,k)$ with $j \in \{0,\ldots,m-1\}$ and $k \in \{1,2,3\}$, it returns the k-th literal of the j-th clause.*
The problem is to decide whether the encoded formula is satisfiable.

The $\mathcal{NEXPTIME}$-completeness for this problem has been proved by Papadimitriou and Yannakakis in [14].

For the reduction of SUCCINCT-3SAT to the scheduling problem, the first crucial step is a transformation of a Boolean circuit B into a DPG.

Each input gate x of B is encoded by two nodes x_f and x_t with output mode PAR. The meaning of the indices here and later is as follows: x_f codes the value *false* for x, while x_t *true* . A NOT gate x is encoded by a graph of four vertices x_f, x_t, v_f, v_t with two edges: (x_f, v_t) and (x_t, v_f). The nodes x_f, x_t represent the input of this gate, while v_f, v_t the output.

Each AND , resp. OR gate is represented by a graph of ten vertices $x_f, x_t, y_f, y_t, p_1, p_2, p_3, p_4, v_f, v_t$ as follows. In both cases, there are edges $(x_f, p_1), (y_f, p_1), (x_t, p_4), (y_t, p_4), (x_f, p_2), (y_t, p_2), (x_t, p_3), (y_f, p_3)$. Furthermore, for an AND gate we add the edges $(p_1, v_f), (p_2, v_f), (p_3, v_f), (p_4, v_t)$, while for an OR gate $(p_1, v_f), (p_2, v_t), (p_3, v_t), (p_4, v_t)$. p_1, p_2, p_3, p_4 are internal nodes to establish the connection between the x and y inputs to the v outputs.

The input and output modes of these subgraphs are defined as follows: For all sources $x \in \{x_f, x_t, y_f, y_t\}$ choose $I(x) = O(x) := \mathsf{ALT}$; for all sinks $v \in \{v_f, v_t\}$ let $I(v) := \mathsf{ALT}$ and $O(v) := \mathsf{PAR}$; finally, for all inner nodes p_i define $I(p_i) := \mathsf{PAR}$ and $O(p_i) := \mathsf{ALT}$.

To encode an arbitrary Boolean circuit B by a DPG $\mathcal{G}$ we code each gate of B separately as described above and draw edges from the output nodes of each gate to the appropriate input nodes.

This encoding achieves the following property. Let $x_1, x_2, \ldots, x_r$ be the input gates of B, and $v_1, v_2, \ldots, v_s$ its output gates. In the coding graph $\mathcal{G}$ there will be sources x_{fi}, x_{ti} for $1 \le i \le r$, and sinks v_{fj}, v_{tj} for $1 \le j \le s$. To understand the functionality assume that to the graph $\mathcal{G}$ additional temporary nodes $x_1, x_2, \ldots, x_r$ are added, each with output mode ALT. x_i will be connected to x_{fi} and x_{ti}. To simplify notation, let $x_i(\mathit{false}) := x_{fi}$ and $x_i(\mathit{true}) := x_{ti}$, etc. Then the following holds: for any input $(b_1, b_2, \ldots, b_r)$, B returns $(c_1, c_2, \ldots, c_s)$ iff there exists a run for $\mathcal{G}$ such that each of the process nodes $x_1(b_1), \ldots, x_r(b_r)$ and $v_1(c_1), \ldots, v_s(c_s)$ has one execution instance, and none of the complementary nodes $x_1(\neg b_1), \ldots, x_r(\neg b_r)$, $v_1(\neg c_1), \ldots, v_s(\neg c_s)$ has any execution instance. As an example see Fig. 9.

Now assume that B is a given input circuit of the SUCCINCT-3SAT problem that codes a Boolean 3CNF formula $\mathcal{F}$ as specified above. We may assume that $\mathcal{F}$ has exactly $m = 2^d$ clauses. Then there will be $n = 2^{d-1}$ variables.

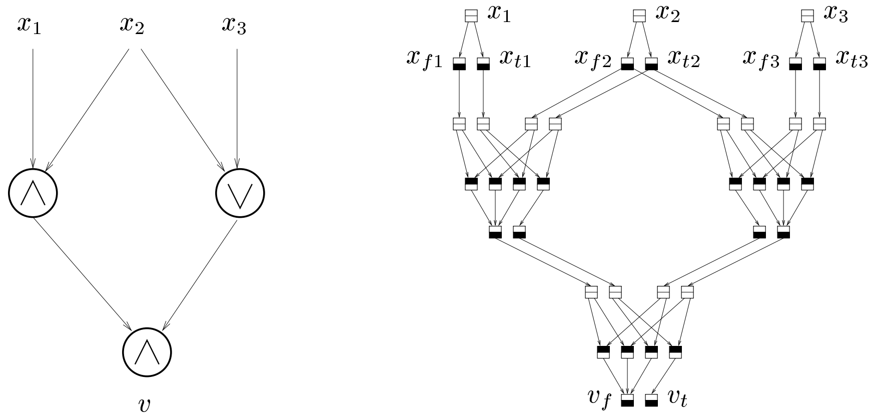

Figure 9: *A Boolean circuit and its DPG with the auxiliary nodes* x_i.

The first step of our reduction is to construct four circuits: A, A_1, A_2, and A_3. The first circuit verifies the syntax of the formula encoded by B. Speaking more formally, A with the input j returns 1 if and only if the given circuit B returns j on exactly three inputs $\boldsymbol{(k,\ B(2,j,k'),\ k'')}$ for all $k \in \{0,1\}$ and $k', k''\{1,2,3\}$. Otherwise A returns 0.

For $k = 1,2,3$, circuits A_k, with the input $\boldsymbol{(j,i)}$, are defined as follows. A_k returns $\mathbf{(1,1)}$ if $B(2,j,k) = i$, $B(1,i,k') = j$ for some k', and $B(0,i,k'') \neq j$ for any k'', with $k', k'' \in \{1,2,3\}$, that means x_i (but not $\neg x_i$) is the k-th literal of the j-th clause. A_k returns $\mathbf{(1,0)}$ if $B(2,j,k) = i$, $B(0,i,k') = j$ for some k', and $B(1,i,k'') \neq j$ for any k'', in other words $\neg x_i$ (but not x_i) is the k-th literal of the j-th clause. If both literals x_i and $\neg x_i$ appear in the j-th clause, i.e. $B(1,i,k')$ and $B(0,i,k'')$ are equal to j for some k', k'', then there are also at least two values $k_1, k2 \in \{1,2,3\}$ such that $B(2,j,k_1) = B(2,j,k_2) = i$. In this case A_k returns $\mathbf{(1,0)}$ if $k = \min\{\ell \in \{1,2,3\} \mid B(2,j,\ell) = i\}$ and A_k returns $\mathbf{(1,1)}$ for all other $k \in \{\ell \in \{1,2,3\} \mid B(2,j,\ell) = i\}$. Finally, in any other case, A_k returns $\mathbf{(0,0)}$. Obviously, these circuits can be easily constructed from given B. Let $\mathcal{A}, \mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$ be the DPGs which denote the encode circuits A, A_1, A_2, A_3. We will call $\mathcal{A}$ the **syntax-verifier**, and $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$ the **index-verifier** graphs.

According to our encoding method, $\mathcal{A}$ has $2d$ sources, and each graph $\mathcal{A}_k$ has $2d + 2(d-1)$ sources, that encode integers j and i, with $0 \leq j \leq m-1$ and $0 \leq i \leq n-1$. Let us call the sources of $\mathcal{A}$ $\mathbf{j_{t1}}, \mathbf{j_{f1}}, \mathbf{j_{t2}}, \mathbf{j_{f2}}, \ldots, \mathbf{j_{t\mathit{d}}}, \mathbf{j_{f\mathit{d}}}$. They encode integer j as follows: If the ℓ-th digit of the binary representation of j is 1 then in a run for $\mathcal{A}$ which represent j, the node $\mathbf{j_{t\ell}}$ has exactly one execution instance, while $\mathbf{j_{f\ell}}$ has no execution at all. If the ℓ-th digit of j is 0 then $\mathbf{j_{f\ell}}$ has exactly one execution instance and $\mathbf{j_{t\ell}}$ has none. This unique instance of either $\mathbf{j_{t\ell}}$ or $\mathbf{j_{f\ell}}$ will be denoted by $t(j,\ell)$.

Similarly, we denote the sources of $\mathcal{A}_k$ by $\mathbf{j_{t1}}, \mathbf{j_{f1}}, \mathbf{j_{t2}}, \mathbf{j_{f2}}, \ldots, \mathbf{j_{t\mathit{d}}}, \mathbf{j_{f\mathit{d}}}$ and $\mathbf{i_{t1}}, \mathbf{i_{f1}}, \ldots, \mathbf{i_{t\mathit{d}-1}}, \mathbf{i_{f\mathit{d}-1}}$, and the executions of these nodes represent integers j and i as above with instances $t(j,\ell)$, resp. $t(i,\ell)$. Note also, that according to the construction method, $\mathcal{A}$ has two sink nodes and each of $\mathcal{A}_k$ has four. Let the number $\boldsymbol{\alpha} := |\mathcal{A}| + |\mathcal{A}_1| + |\mathcal{A}_2| + |\mathcal{A}_3|$ count the number of nodes of these graphs.

To generate the representation of the input values i and j some additional structures will be needed. First, let us introduce the **line DPG** of length $\ell \geq 1$, which consists of ℓ vertices $v_1, v_2, \ldots, v_\ell$ and $\ell - 1$ edges: $(v_1, v_2) \ldots (v_{\ell-1}, v_\ell)$. Their input and output modes are inessential, we set $I \equiv O \equiv \mathsf{PAR}$. The second DPG, called a **chain** of length t for $t \geq 0$, consists of $3 \cdot t + 1$ vertices $p_0, p_1, \ldots, p_t, l_1, \ldots, l_t, r_1, \ldots, r_t$, and edges $(p_i, l_i), (p_i, r_i), (l_i, p_{i-1}), (r_i, p_{i-1})$ for $i = t, \ldots, 1$. Note, that for $t = 0$, the chain contains just one node. Moreover, choose $I \equiv \mathsf{ALT}$ and $O(x) := \mathsf{PAR}$ for all nodes of the chain graph except the sink, for which $O(p_0) := \mathsf{ALT}$.

Furthermore, for $h \geq 1$ we define the **counting graph** of height h consisting of one chain of length h, called the **main** chain, and for every $t \in \{h-1, \ldots, 0\}$ of pairs of chains, a **left** and a **right** one, each of length t, . For every such t add one edge from l_{t+1} of the main chain to the source of the left chain of length t, and one edge from r_{t+1} of the main chain to the source of the right chain of length t. Finally, we modify the whole structure as follows: In the main chain, for all $i \in \{1, \ldots, h\}$ every node l_i (resp. every node r_i) is replaced by a line graph of length $\alpha + 1$. Input connections of l_i (resp. r_i) are driven to the source of the line graph, while output connections run from its sink. An example of a counting graph with $h = 2$ and $\alpha = 2$ is given in Fig. 10.

As special incarnation, define the **clause-index graph** as a counting graph of height d and the **variable-index graph** as a counting graph of height $d-1$ (recall that $m = 2^d$). For the clause-index graph let us denote the sink of the main chain by $\mathbf{C}$ and the remaining sinks by $\mathbf{j_{t\ell}}$,$\mathbf{j_{f\ell}}$, with $1 \leq \ell \leq d$. More precisely, let $\mathbf{j_{t\ell}}$ denote the sink of the right chain and $\mathbf{j_{f\ell}}$ the sink of the left chain of length $\ell-1$. An execution instance of the sink of the main chain $\mathbf{C}$ of the clause-index graph represents the clause the index of which is represented by the execution pattern of the sinks of the other chains (for each ℓ only one sink of the two chains of length ℓ will be executed, either the left or the right one). Similarly, we denote by $\mathbf{X}$ the sink of the main chain of the variable-index graph, and for $\ell = 1, \ldots, d-1$ by $\mathbf{i_{t\ell}}$ (resp. $\mathbf{i_{f\ell}}$) the sink of the right (resp. left) chain of length $\ell-1$.

Finally, we introduce the **clause-index-duplicator graph** and the **variable-index-duplicator graph** that are slightly modified variants of the corresponding counting graphs. Let $\boldsymbol{R_k}$ be a DPG consisting of nodes $u, p_1, \ldots, p_k, v$ and edges $(u, p_1), \ldots, (u, p_k)$ and $(p_1, v), \ldots, (p_k, v)$. The output mode of u is PAR, and the output modes of the nodes $p_1, \ldots, p_k, v$ is ALT. Further we choose $I(x) = \mathsf{ALT}$ for all nodes of R_k. The clause-index-duplicator graph is a clause-index graph in which $2d$ copies of R_3 are added: each of $\mathbf{j_{t1}}, \mathbf{j_{f1}}, \ldots, \mathbf{j_{t\mathit{d}}}, \mathbf{j_{f\mathit{d}}}$ is connected to the source of its own copy of R_3. In this graph rename the vertices by simply "shifting down" the names $\mathbf{j_{t\ell}}$ and $\mathbf{j_{f\ell}}$ to the sinks of the copies of the R_3 graph.

A variable-index-duplicator is obtained in a similar way by connecting each node $\mathbf{i_{t1}}, \mathbf{i_{f1}}, \ldots, \mathbf{i_{t\mathit{d}-1}}, \mathbf{i_{f\mathit{d}-1}}$ to one copy of the R_6 graph. In the graph obtained this way we also "shift down" the names $\mathbf{i_{t1}}, \mathbf{i_{f1}}, \ldots, \mathbf{i_{t\mathit{d}-1}}, \mathbf{i_{f\mathit{d}-1}}$ of the old to the new sinks.

The total construction consists of two disjoint graphs made up of the subgraphs described above. The first one $\mathcal{G}_I$ will be responsible for checking whether the input circuit B generates a Boolean formula according to the syntax of SUCCINCT-3SAT. The second graph $\mathcal{G}_{II}$ checks whether the encoded formula is satisfiable.

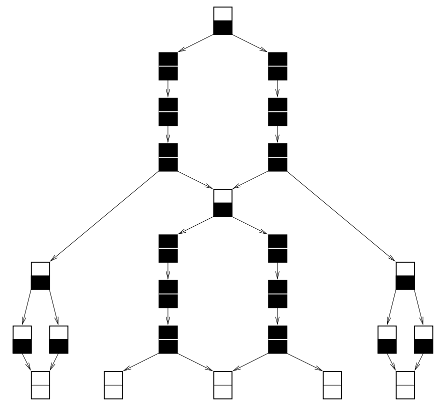

Figure 10: *The counting graph for $h = 2$ and $\alpha = 2$.*

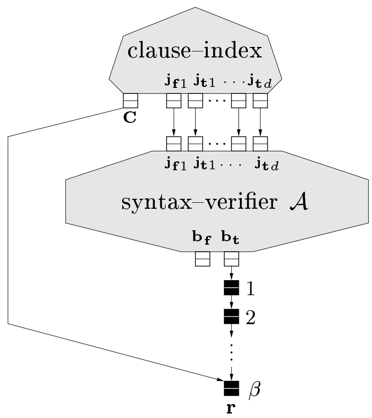

Figure 11: *DPG $\mathcal{G}_I$ that meets the deadline iff B generates a correct encoding.*

Part I. To construct a dynamic process graph $\mathcal{G}_I$ that verifies the syntax of SUCCINCT-3SAT we will use a clause-index graph, a syntax-verifier $\mathcal{A}$, and a line graph of length β, where $\beta \geq 1$ is a parameter of this part (which will be fixed later depending on the size of $\mathcal{G}_{II}$ the graph of the second part). Connecting these subgraphs we draw edges from the nodes $\mathbf{j_{t1}}, \mathbf{j_{f1}}, \ldots, \mathbf{j_{t\mathit{d}}}, \mathbf{j_{f\mathit{d}}}$ of the clause-index graph to the corresponding sources in $\mathcal{A}$. Recall that $\mathcal{A}$ has two sink nodes – call them $\mathbf{b_f}$ and $\mathbf{b_t}$ – that encode the output bit of the circuit A. This bit equals 1 for input j iff circuit B correctly encodes the j-th clause.

To complete the construction of $\mathcal{G}_I$ we draw one edge from $\mathbf{b_t}$ to the source and one from $\mathbf{C}$ to the sink of the line graph. The sink of the line graph will be denoted by $\mathbf{r}$ (see Fig. 11).

To guarantee a correct encoding of the integers $j \in \{0, \ldots, m-1\}$ we like all tasks which correspond to this integer, i.e. $t(j, \ell)$, to be executed by the same processor. These execution instances $\mathbf{j_{t1}}, \mathbf{j_{f1}}, \ldots, \mathbf{j_{t\mathit{d}}}, \mathbf{j_{f\mathit{d}}}$ can be forced appropriately by a suitable communication delay and deadline. Defining

$$\boldsymbol{D_c} \;:=\; 1 + d \cdot (\alpha + 2) + \sum_{\ell=0}^{d-1} (2\ell + 1) \qquad \text{and} \qquad \boldsymbol{T_1^*(\beta)} \;:=\; D_c + |\mathcal{A}| + \beta.$$

for any integer β we can prove the following relation.

Lemma 5 *For $\beta \geq 1$ and any communication delay $\delta_1 \geq D_c + |\mathcal{A}|$, $\mathcal{G}_I$ can be scheduled with deadline $T_1^*(\beta)$ iff B encodes a 3-CNF formula $\mathcal{F}$, where each literal appears exactly three times.*

Part II. The construction of the essential DPG $\mathcal{G}_{II}$ is technically more involved than that for the syntax checker. A clause-index-duplicator and a variable-index-duplicator graph will be used. Moreover, we will add the DPGs $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$, and some auxiliary nodes.

The construction starts by drawing edges between the clause-index-duplicator and the $\mathcal{A}_k$ graphs in a similar way as previously, i.e. connecting each of the nodes $\mathbf{j_{t1}}, \mathbf{j_{f1}}, \ldots, \mathbf{j_{t\mathit{d}}}, \mathbf{j_{f\mathit{d}}}$ with its counterparts in $\mathcal{A}_1, \mathcal{A}_2$ and $\mathcal{A}_3$. We connect also each of the nodes $\mathbf{i_{t1}}, \mathbf{i_{f1}}, \ldots, \mathbf{i_{t\mathit{d}-1}}, \mathbf{i_{f\mathit{d}-1}}$ of the variable-index-duplicator with appropriate $\mathbf{i_{t\ell}}, \mathbf{i_{f\ell}}$ nodes of $\mathcal{A}_1, \mathcal{A}_2$ and $\mathcal{A}_3$. Recall that $\mathcal{A}_k$ has four sink nodes:

- the first pair, call them $\mathbf{a_f}$ and $\mathbf{a_t}$, encode the first output bit of the circuit A_k that equals to 1 if for the input (j,i), x_i or $\neg x_i$ is the k-th literal of the j-th clause;
- the second pair of $\mathcal{A}_k$, let us call them $\mathbf{b_f}$ and $\mathbf{b_t}$, encode the second output bit of A_k that equals to 1 if x_i is the k-th literal of the j-th clause and 0 elsewhere.

The following fragment will be responsible for assigning *true* or *false* to the particular variable. Let us introduce two new nodes $\mathbf{T}$ and $\mathbf{F}$ which are connected to $\mathbf{X}$ via $(\mathbf{X}, \mathbf{T})$ and $(\mathbf{X}, \mathbf{F})$. Remember that the output mode of $\mathbf{X}$ is ALT which enforces that for each execution instance of $\mathbf{X}$ there exists exactly one edge connecting this task of $\mathbf{X}$ either with an execution of $\mathbf{T}$ or with an execution of $\mathbf{F}$. This guarantees that to each variable x exactly one value – either *true* or *false* – is assigned. To transfer this value to all literals x and $\neg x$ that appear in the formula we define $O(\mathbf{T}) = O(\mathbf{F}) := \mathsf{PAR}$ and connect these nodes to the sources of four copies of the R_3 graph, two for each. One of the sinks of the two R_3 graphs connected with $\mathbf{T}$ will be called $\mathbf{x_t}$, and the other one $\neg\mathbf{x_f}$. Analogously call the sinks of the R_3 graphs connected with $\mathbf{F}$ $\neg\mathbf{x_t}$ and $\mathbf{x_f}$. The output mode of these sinks is ALT.

Next, for each graph $\mathcal{A}_k$, with $1 \leq k \leq 3$, new nodes $\mathbf{v}, \neg\mathbf{v}, \mathbf{v_t}, \mathbf{v_f}, \neg\mathbf{v_t}, \neg\mathbf{v_f}, \mathbf{r}, \mathbf{s}$ with input mode PAR are introduced, and a node $\mathbf{val}$ with input mode ALT. The output mode of all these nodes is ALT. We connect these nodes by the edges $(\mathbf{a_t}, \neg\mathbf{v})$, $(\mathbf{a_t}, \mathbf{v})$, $(\mathbf{b_f}, \neg\mathbf{v})$, $(\mathbf{b_t}, \mathbf{v})$ and $(\mathbf{v}, \mathbf{v_f})$, $(\mathbf{x_f}, \mathbf{v_f})$, $(\mathbf{v}, \mathbf{v_t})$, $(\mathbf{x_t}, \mathbf{v_t})$ and symmetrically $(\neg\mathbf{v}, \neg\mathbf{v_f})$, $(\neg\mathbf{x_f}, \neg\mathbf{v_f})$, $(\neg\mathbf{v}, \neg\mathbf{v_t})$, $(\neg\mathbf{x_t}, \neg\mathbf{v_t})$. Finally, the following edges $(\neg\mathbf{v_t}, \mathbf{val})$, $(\mathbf{v_t}, \mathbf{val})$, $(\mathbf{val}, \mathbf{r})$, $(\mathbf{val}, \mathbf{s})$ are added. The whole construction gets completed by drawing edges from the sink $\mathbf{C}$ of the clause-index graph (remember, the output mode of $\mathbf{C}$ is ALT) to the first, second and third copy of the node $\mathbf{r}$ (see Fig. 12).

The communication delay and the deadline of the final graph are defined as

$$\boldsymbol{\delta} \;:=\; D'_c + D'_v + \max_{1 \leq k \leq 3} \{|\mathcal{A}_k|\} \qquad \text{and} \qquad \boldsymbol{T^*} \;:=\; \delta + 8\,,$$

where $D'_c \;:=\; D_c + 2d$, $D_v \;:=\; (d-1) \cdot \alpha + \sum_{\ell=0}^{d-1} (2\ell + 1)$ and $D'_v \;:=\; D_v + 2(d-1)$.

Lemma 6 *$\mathcal{G}_{II}$ can be scheduled by deadline T^* with respect to communication delay δ iff the formula $\mathcal{F}$ encoded by B is satisfiable.*

To complete the proof of Theorem 5 let $\delta_1 := \delta$ and $\beta := T^* - (D_c + |\mathcal{A}| - 1)$. Note that δ_1 fulfills the inequality as required by Lemma 5. Then Lemma 5 and 6 imply

Lemma 7 *The union of $\mathcal{G}_I$ and $\mathcal{G}_{II}$ can be scheduled within the deadline T^* given communication delay δ iff the corresponding* SUCCINCT-3SAT *problem has a positive solution.*

∎

From Lemma 1 and Lemma 4 one can deduce that the nondeterministic exponential lower time bound is sharp.

Theorem 6 *An optimal schedul for dynamic process graphs can be computed in $\mathcal{NEXPTIME}$.*

If one restricts the DPGS problem to specific combinations of input or output modes then in most cases the computational complexity decreases significantly. Below we list the results of our investigations. The proofs of these hardness results and the algorithms to establish the upper bounds can be found in [8].

This gives a precise complexity theoretic characterization of scheduling dynamic process graphs, except for the case of output mode PAR, where lower and upper bounds do not match.

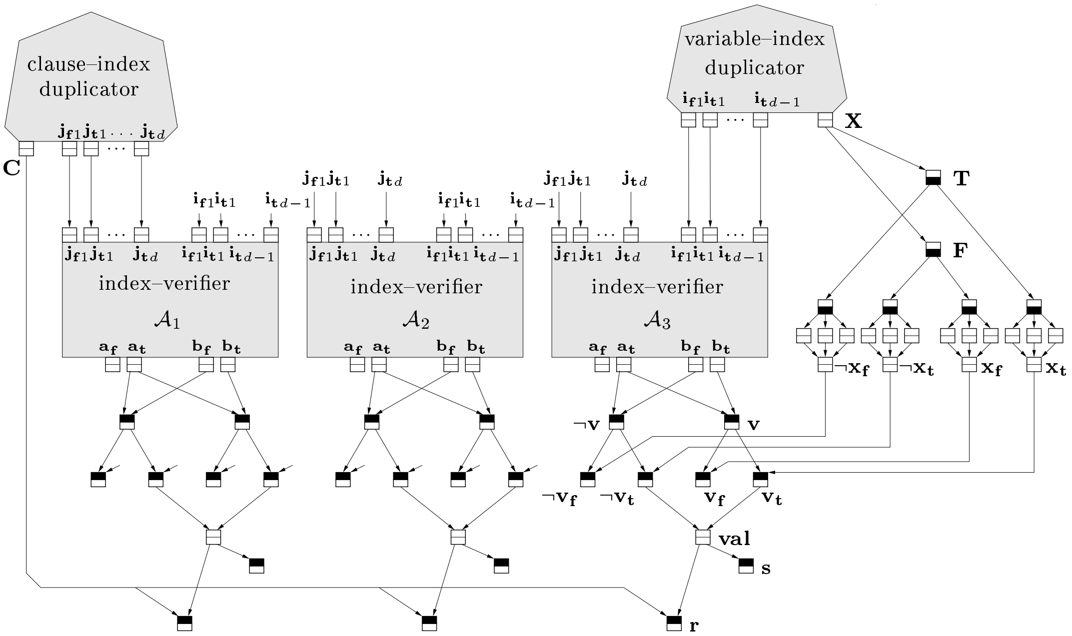

Figure 12: *The construction of $\mathcal{G}_{II}$*

input mode	output mode	complexity	problem for reduction
ALT	ALT	$\mathcal{NLOGSPACE}$-complete	Directed Graph Reachability Problem
ALT	PAR	$\mathcal{NLOGSPACE}$-complete	complement of DGR Problem
ALT	ALT, PAR	$\mathcal{P}$-complete	Circuit Value Problem
PAR	ALT, PAR or PAR or ALT	$\mathcal{NP}$-complete	Scheduling Problem with
ALT, PAR	ALT	$\mathcal{NP}$-complete	Communication Delay for in-Trees
ALT, PAR	PAR	$\mathcal{BH}_2$-hard	SAT-UNSAT Problem
ALT, PAR	ALT, PAR	$\mathcal{NEXPTIME}$-complete	SUCCINCT-3SAT

Figure 13: *The different variants of scheduling DPGs*

5 Conclusion

We have defined a dynamic model for scheduling process graphs. These dynamic process graphs allow a compact representation of typical distributed programs written in OCCAM or Ada style. We are not aware of another framework with a similar expressive power. Restricting the input and output mode of nodes different degrees of concurrency can be modelled. An open quesition is to understand in more detail the expressive power of the different combinations of ALT and PAR modes at input and output. As mentioned in the beginning there are some relations to Petri nets. Among the four possible combinations of input and output modes, for the execution problem it suffices to consider only two of them: [ALT | PAR] and [PAR | ALT]. The reason is that a pairing gives the other two.

We have analysed with respect to the degree of concurrency how difficult it is to decide whether a dynamic process graph can be executed and to construct optimal schedules. An almost complete exact characterization could be given. Only for the scheduling problem with output mode restricted to PAR there remains a gap.

For the general case, we have shown an exponential complexity jump when scheduling process graphs in the dynamic setting. This implies that our compact dynamic representation is quite effective. Constructing optimal schedules of standard graphs cannot be done efficiently, but at least the problem can be approximated to a certain factor by simple algorithmic methods. This even holds when one takes communication delays into account. We do not have any nontrivial approximation results for scheduling dynamic process graphs.

References

[1] A. Burns, *Programming in Occam 2,* Addison-Wesley Publishing Company, 1988.

[2] M. Dyer and A. Frieze, *Planar 3DM Is NP-Complete*, J. Algorithms 7, 1986, 174-184.

[3] H. El-Rewini and H. H. Ali, *Static Scheduling of Conditional Branches in Parallel Programs,* J. Par. Distrib. Comput. 24, 1995, 41-54.

[4] H. Galperin, A. Wigderson, Succinct Representations of Graphs, Information and Control, 56, 1983, 183-198.

[5] M. R. Garey and D. S. Johnson, *Computers and Intractability, A Guide To the Theory of NP-Completeness,* Freeman 1979.

[6] R. Greenlaw, H. J. Hoover and W. L. Ruzzo, *Limits to Parallel Computation – P-Completeness Theory*, Oxford University Press, 1994.

[7] S. Ha, E. Lee, *Compile-time Scheduling and Assignment of Data-flow Program Graphs with Data-dependent Iteration*, IEEE Trans. Computers 40, 1991, 1225-1238.

[8] A. Jakoby, M. Liśkiewicz, R. Reischuk, *Scheduling Dynamic Graphs,* Technischer Bericht, Institut für Theoretische Informatik, Med. Universität zu Lübeck, 1998.

[9] A. Jakoby and R. Reischuk, *The Complexity of Scheduling Problems with Communication Delay for Trees,* Proc. 3. Scandinavian Workshop on Algorithm Theory SWAT'92, 1992, 165 - 177.

[10] H. Jung, L. Kirousis, P. Spirakis, *Lower Bounds and Efficient Algorithms for Multiprocessor Scheduling of DAGs with Communication Delays,* Proc. 1. SPAA, 1989, 254 - 264.

[11] R. Kieckhafer, *Fault-Tolerant Real-Time Task Scheduling in the MAFT Distributed System*, 22. Hawaii Int. Conf. on System Science, 1989, 145-151.

[12] R. Kieckhafer, C. Walter, A. Finn, P. Thambidurai, *The MAFT Architecture For Distributed Fault-Tolerance*, IEEE Trans. Computers, April 1988, 398-405.

[13] T. Lengauer, K. Wagner, *The Correlation between the Complexities of the Nonhierarchical and Hierarchical Versions of Graph Problems,* J. CSS 44, 1992, 63-93.

[14] C. Papadimitriou and M. Yannakakis, *A Note on Succinct Representations of Graphs*, Information and Control, 71, 1986, 181-185.

[15] C. Papadimitriou and M. Yannakakis, *Towards an Architecture-Independent Analysis of Parallel Algorithms,* Proc. 20. STOC, 1988, 510-513, see also SIAM J. Comput. 19, 1990, 322-328.

[16] B. Veltman, *Multiprocessor Scheduling with Communication Delays,* Ph.D. Thesis, University of Technology Eindhoven, Department of Computer Science, Eindhoven, The Netherlands, 1993.