Well-Formed Generalized Task Graphs

M. Ghodsi and K. Kant

Computer Science Department
Pennsylvania State University
University Park, PA 16802

Abstract

The generalized task graphs are the classical fork-join task graphs augmented with other types of nodes, namely branch and merge nodes, and a new node type called *max* node which greatly increases the modeling power of the task graph. The generalized task graphs can be used to model a richer class of parallel computations including non-deterministic programs. Such task graphs are prone to problems such as deadlocks, unboundedness, and unsafeness. Therefore, it is necessary to have techniques for the detection of such problems. With this goal in mind and by viewing the generalized task graphs as high level Petri nets, we define the notion of well-formedness of task graphs. We then provide the necessary and sufficient conditions under which a generalized task graph is well-formed. We also present a method for checking these conditions based on the structural analysis of task graphs. This method is generally less expensive than the straight forward techniques based on the reachability analysis.

Keywords: model of computations, task graphs, Petri nets, liveness, safeness.

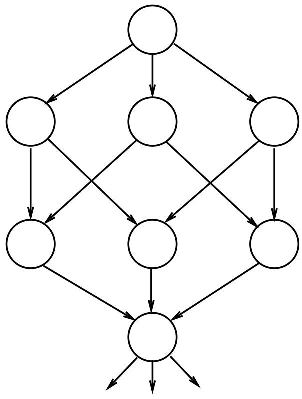


Figure 1: A 2-out-of-3 operation implemented using a max node.

1 Introduction

Graph-based models have been frequently used to represent parallel computations [1, 9, 12]. A parallel computation, based on its logical structure, is decomposed into a number of tasks with certain ordering constraints between them and is usually described by a task graph in which nodes represent the tasks and the arcs represent the precedence relations between the tasks. A task graph may also be used to model the details of the processing stages of a data flow program [4, 10], or to represent the dependencies between separate programs in a large software system.

The *classical task graphs* are acyclic graphs which contain only the well-known *fork* and *join* nodes. To model loops and conditional branches, classical task graphs have been augmented with other types of nodes, namely *branch* and *merge* nodes [1]. We call this class of task graphs as the *extended task graphs* (ETG).

The extended task graphs are inadequate for modeling systems with complex non-deterministic behavior. One example is the systems where several solution approaches to the problem are tried concurrently, and the first successful solution is accepted and the other computations are aborted. Many algorithms in the areas of game theory and the artificial intelligence have this characteristic. Another example is the use of task graphs as the fault diagrams in the reliability modeling of systems. In the fault trees, there are elements, such as k-out-of-m components, that cannot be modeled by the extended task graphs.

The interest in modeling more complex systems has led to the introduction of new nods types in the task graphs. Sahner and Trivedi [15] have used k-out-of-m nodes in the context of fault trees. Kant [8] has introduced a new node, called 'max' node, to be used in the general acyclic task graphs. Figure 1.b which is the implementation of a 2-out-of-3 operation using a max node, illustrates the function of this node. Tasks A, B, and C are created simultaneously. The max node D, labeled with '@', is initiated whenever any two of the three tasks are completed. An additional feature of this node is that it will choose a different path out of D depending on which task pair finished first.

In this paper, we extend the power of the *max* node defined in [8] so that it can be used in a more general setting. In particular, the we assume that the max node has the power to *abort all* the remaining *competing computations* after one of them is completed. This characteristic allows the max nodes to be used in the cyclic task graphs as well. The extended task graphs augmented with the max nodes are called the *generalized task graphs* (GTG).

The modeling powers of different classes of task graphs can be evaluated by comparing the modeling powers of the equivalent classes of Petri nets [13, 14]. The equivalent Petri nets for the classical task graphs fall into the category of *decision-free* Petri nets [3] (also called marked graphs.) A Petri net is said to be decision-free if the number of the input transitions and the output transitions of every place is one. The extended task graphs are isomorphic to *free-choice* Petri nets [11] which is a more powerful class for which many results are known. In a free-choice Petri net, if a place has more than one output transitions, it must be the only input place of those transitions [6]. As we shall see later in the paper, the Petri net representation for a generalized task graph requires inhibitor arcs (or equivalent extensions), which makes the GTG's more powerful model than the extended task graphs.

A generalized task graphs may not represent a meaningful parallel computation, because (1) the computation may suffers from deadlock, (2) there may be a task in the task graph that is never initiated, (3) the computation may create an unbounded number of synchronization messages forwarded to a task, and (4) a task may be reinstantiated while it is already being processed (this property may be acceptable in some cases.) We call a task graph *well-formed* or *legal* if it is free of these problems. The analysis of a generalized task graph for the existence of these problems is the subject of this paper. We are interested in providing techniques based on which the well-formedness of a GTG can be verified.

One straight forward solution to this problem is doing the reachability analysis [13] of the equivalent Petri net model of a GTG and detecting the possible problems that may exist. This approach can be quite expensive, because the reachability tree can be very large. The other techniques, discussed in this paper, are those that are based on the structural properties of the task graphs. Unfortunately, both approaches become infeasible for very large task graphs because the liveness problem for the free-choice Petri nets (and thus for the extended task graphs) is NP-complete [7]. However, compared to the reachability analysis, the structural analysis is generally less expensive, because its complexity does not depend on the dynamics of the task graph, but rather is a function of the number of nodes, arcs, and the combination of node types.

The problem of verifying the legality of a task graph was first considered in [2] where the authors present a detailed algorithm to check the well-formedness of the acyclic extended task graphs. The legality of the general forms of the extended task graphs can be verified using Hack's results on the liveness and the safeness of the free-choice Petri nets [6]. The analysis of the generalized task graphs was also considered in [8] in which a new approach based on the algebraic expression of the flow rates was proposed. However, this method is only applicable to the acyclic task graphs where max nodes are used in a somewhat restricted fashion.

The results that we present here for the analysis of the GTG's are applicable to cyclic task graphs as well as more general topologies for the max nodes. We follow an approach similar to that of Hack's and prove similar results, even though the GTG's are more powerful than the free-choice Petri nets. This is achieved by viewing the GTG's as high level Petri nets rather than dealing with their detailed Petri net models.

2 Basic Definitions and Terminology

A generalized task graph (GTG) or simply a task graph (TG) is a quadruple $\{\Theta, \prec, IL, OL\}$ where $\Theta = \{t_1, \ldots, t_K\}$ represents the set of tasks, $\prec$ is the precedence relation defined on Θ, and IL and OL are the functions: $\Theta \rightarrow \{$ '*', '+', '@', $\}$ that represent the input and output labels of the tasks respectively. With obvious interpretations, we use the notations $isucc(t)$ and $ipred(t)$ to denote the set of tasks that are the immediate successors and the immediate predecessors of a task t respectively. Similarly, $pred(t)$ and $succ(t)$ are used for the set of predecessors and successors of t. We assume that t_1 is a predecessor to all the tasks in the task graph and call it the *initial* task. Similarly, t_K is assumed to be is a successor to all tasks and is called the *terminal* task. The initial and the terminal tasks are the first and the last tasks that are enabled in the TG respectively.

In the graph representation, Θ is the node set and E is the set of arcs. E includes two additional arcs, namely the *initial arc* a_s which is the input arc to the initial task and a *terminal arc* a_e which is the output arc for the terminal task. A task is called *simple* if it has only one incoming and one outgoing arc.

A cycle C in the TG, can be uniquely identified by a *backward* arc b and two non-simple tasks, called the *head* and the *tail* tasks of C, that are connected via b.

A task graph can also be viewed as a high level Petri net where the tasks behave like transitions with certain firing rules and the arcs are similar to places that hold tokens. With this in mind, we shall freely use Petri net terminology throughout the paper. We assume that the reader is familiar with these notations.

The execution of a task graph starts with a token at its initial arc and terminates when a token arrives at its terminal arc. A task t becomes *enabled* when, according to its input operation, there are enough tokens in its input arcs $*t$. It then *fires* at which time it removes a number of tokens from its input arcs (and possibly from other arcs as well) and, based on its output operation, puts a number of tokens in its output arcs $t*$. An enabled task can be disabled if one of its enabling tokens is removed.

The operations performed by an enabled task t is represented by its input and output labels. The labels '*'and '+'are used for the fork/join and the branch/merge operations respectively. The firing rules of the tasks with these labels, referred to by $\frac{*}{\cdot}$, $\frac{+}{\cdot}$, $\frac{\cdot}{*}$, and $\frac{\cdot}{+}$ tasks in short, are illustrated in the example of Figure 2 whose equivalent timed Petri net model is also shown in Figure 3. In the figures for the task graphs, we use bars to represent the tasks as opposed to the normally used circles to emphasize our viewing of the task graphs as high-level Petri nets.

We now define the firing rules for a node t with $IL(t)/OL(t)$ = '@'. The nodes with $IL(t)$ = '@' are called *max* nodes. The output label of a max node can be ' $*$ ', ' $+$ ', or '@'. But, a node t with $OL(t)$ = '@' can only have the input label of '@'. We first illustrate the function of a node t with $IL(t) = OL(t)$ = '@' using the example of Figure 4. In this example, t_5 represents a computation C that must finish by some deadline. The deadline is represented by task t_4. If the deadline expires before C finishes, we can have an exceptional situation which must be taken care of. In this case, after t_6 fires, computation C is aborted and task t_7 is initiated. If C meets the deadline, i.e., task t_5 finishes before task t_4, then the control goes to task t_8 and task t_4 is aborted.

In general, corresponding to a max node t, there is a set of *competing computations*, each of which may be represented by a set of tasks, that may also communicate with one another some points during their executions. Node t fires as soon as one of these computations finishes and aborts the rest. If $OL(t)$ = '@', then a different path is chosen out of t depending on the computation that completes first. For this case, therefore, we assume that the number of the input and output arcs of t is equal.

We now formalize the definition of the max nodes, also referred to as $\frac{@}{\cdot}$ and $\frac{@}{@}$ nodes in

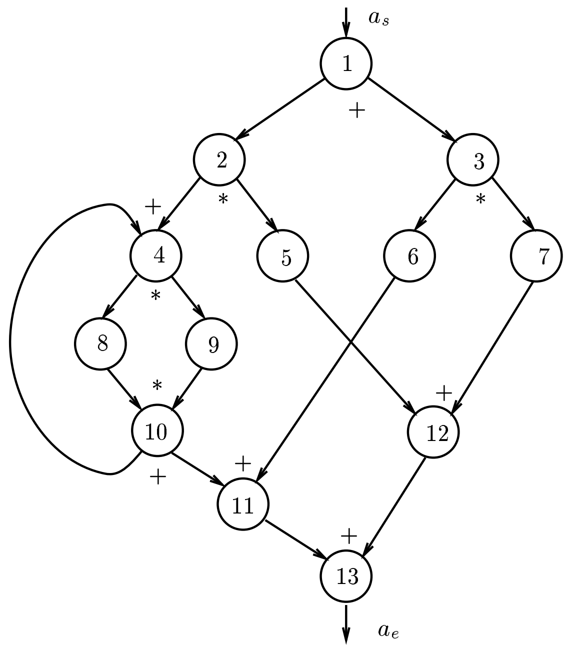


Figure 2: An extended task graph.

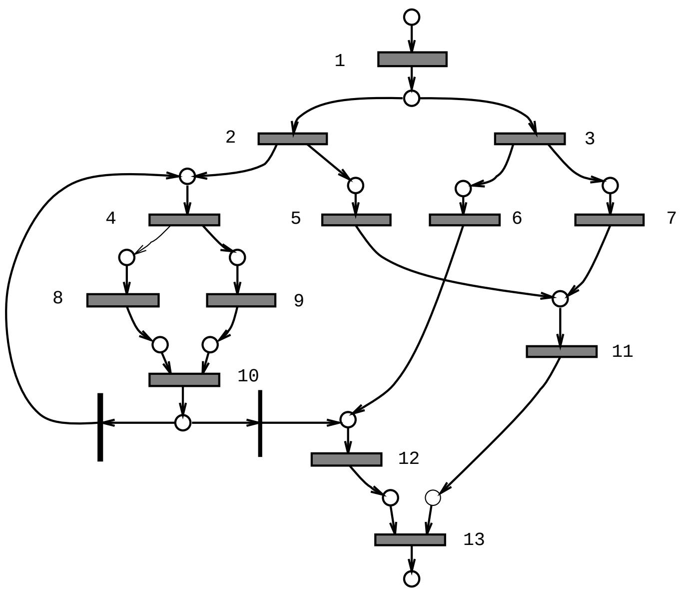


Figure 3: The Petri net representation of the task graph in Figure 2.

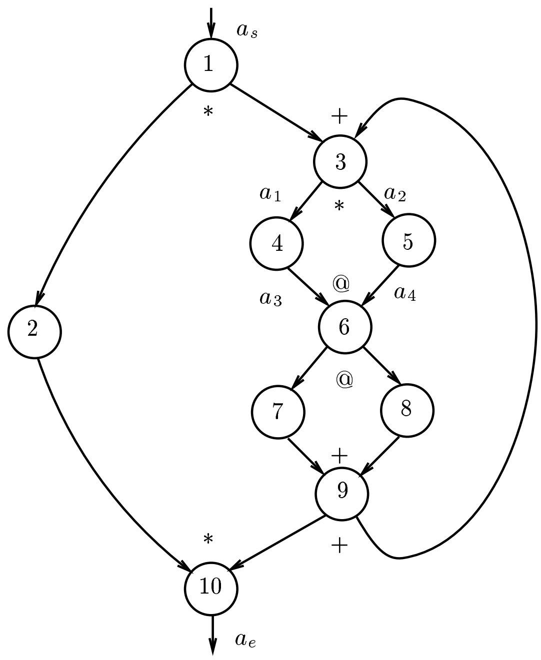


Figure 4: A generalized task graph.

short, and their firing rules. A max node t, with $IL(t) =$ '@', is defined to have the following properties:

1. If $OL(t) =$ '@' then $|t*| = |{*t}| > 1$.

2. A set of arcs is associated with t which is called its *enabling set* and is denoted as ${}^{@}t$. Informally, ${}^{@}t$, which depends on the structure of the task graph, is the set of arcs in the competing computations that end in the firing of t. A formal definition of the enabling sets and a method for their construction will be given in the next section.

3. If $OL(t) =$ '@' then there exists a one-to-one function g_t that maps an input arc $a_i \in {*t}$ to a unique output arc $a_o \in t^*$ and visa versa, i.e., $g_t(a_i) = a_o$ and $g_t(a_o) = a_i$.

4. Task t is enabled immediately after one of its input arcs, say a_i, receives a token. After t fires, *all* of the tokens in ${}^{@}t$ are removed, i.e., other competing computations that are still running are aborted. One token is put in the output arc $g_t(a_i)$ if $OL(t) =$ '@'. The output behavior of t for other output labels has been described before.

Note that the proper function of a max node in the cyclic task graphs, requires the removal of tokens from its enabling set after it fires and, hence aborting some tasks. To illustrate this, consider the example of Figure 4 again where the arcs that are included in the enabling set of the max node t_6 have been identified by black dots. Suppose that t_6 fires as a result of t_5 finishing earlier than t_4, but it does not abort t_4. In the next cycle, this may cause enabling of t_4 again while it is being processed. This would create confusion for the firing of t_6 specially if t_4 finishes first in this cycle, because it is not clear if t_6 should fire or "ignore" the token out

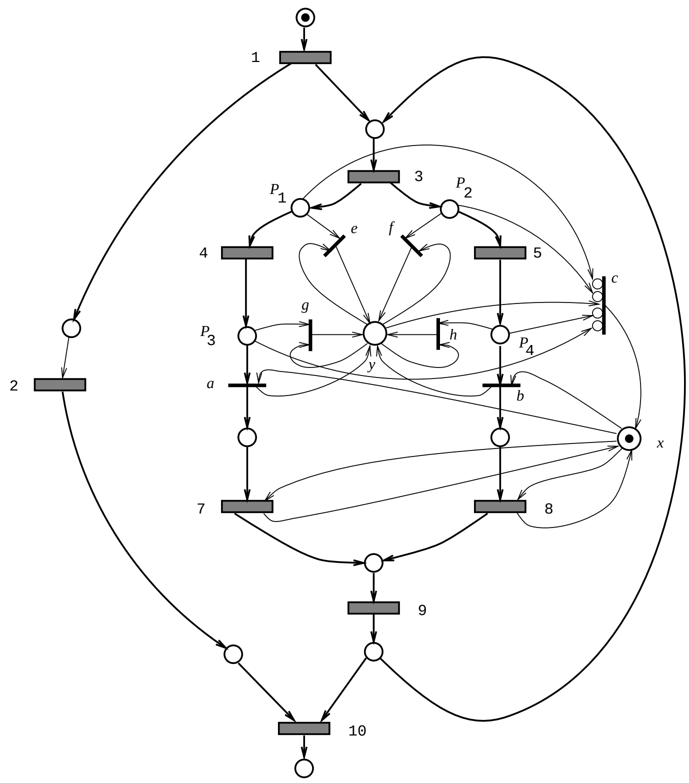


Figure 5: The Petri net representation for the task graph of Figure 4.

t_4. In the acyclic task graphs, however, it is possible for a max node t to ignore other incoming tokens after it fires. This is the firing rule that has been chosen in [8].

The Petri net model for the task graph of Figure 4 is shown in Figure 5. Place x is initially marked to ensure that only one of the transitions a or b can fire. This models the firing of the max node in the task graph. After the firing of a or b, place y gets marked which causes the removal of the remaining token from one of the places in $P = \{p_1, p_2, p_3, p_4\}$ through the firing of transitions e, f, g, or h. These places correspond to the arcs in $^{@}t_6$, namely a_1, a_2, a_3, and a_4. Notice that the firing of transitions 6 or 7 must be put on hold, by making x as an additional input place for both, until the remaining tokens from the places in P are all removed. This is essential, because otherwise, the new tokens that can be injected back into P through the cycle in the net can also be removed from P. Therefore, x needs to receive the token in y immediately after all tokens in P has been removed. This operation, which requires the test for zero token, can only be done in the Petri nets using the inhibitor arcs or other equivalent extensions. Transition c does exactly this; it fires after y is marked and P is blank. This is a typical operation that must be implemented in the Petri nets corresponding to the GTG's with max node which proves the claim that we made earlier that the equivalent Petri net models of the GTG's requires the use of inhibitor arcs.

Definition 1 (liveness) A task t is said to be *live* if there exists a marking reachable from M_0 which can fire t; otherwise t is *dead*. [1] A TG is live iff at any reachable marking from M_0

[1]Note that this definition is different from that for the liveness of transitions in Petri nets. In Petri nets, a transition t is live iff for every marking M reachable from M_0, there exists a firing sequence that enables t.

there exists a sequence of firing of tasks that enables the terminal task, i.e., the task graph always terminates.

Note that the liveness of a TG is not equivalent to the liveness of its terminal task.

It is also possible, in practice, that a computation represented by a GTG is initiated repeatedly many times. Because of this, the *boundness* property of a TG is very important. Informally, a TG is bounded if in its repeated execution, the number of messages (tokens) in every arc remains bounded.

We now define well-formed task graphs. Here, we consider the safeness as a desired property. In most cases, an unsafe TG is unbounded. Also, we do not allow multiple copies of the same task to be processed simultaneously which is another potential problem of unsafe task graphs. There are circumstances, however, that the concurrent execution of the same task makes sense and is not considered illegal, but these cases are not dealt with in this paper.

Definition 2 (well-formed task graph) A task graph is well-formed (legal) iff

1. it is live, i.e, it always terminates,
2. it is safe, and
3. all of its tasks are live.

Figure 6 shows two illegal task graphs.

We next attempt to characterize the well-formedness of the GTG's by examining the properties of the max nodes. As mentioned earlier, max node is a very powerful node. It is also rather difficult to handle since the firing of a max node t may involve the removal of tokens not only from the input arcs of t but from other arcs that are in its enabling set as well. This complication is reflected in the properties and results on the well-formedness of GTG's vis-a-vis of that of the extended task graphs.

3 Properties of max nodes

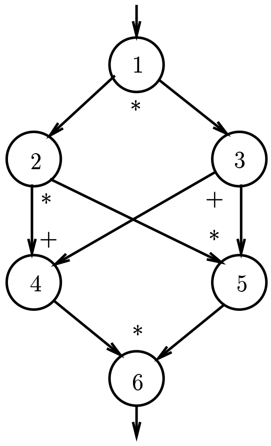
We start with the characterization of the *enabling set* of a max node t, denoted as ${}^{@}t$, which is the set of arcs from which the tokens must be removed when t fires. This set is defined as follows:

Definition 3 (the enabling set: ${}^{@}t$)
For a max node t we define its enabling set ${}^{@}t$ as follows:

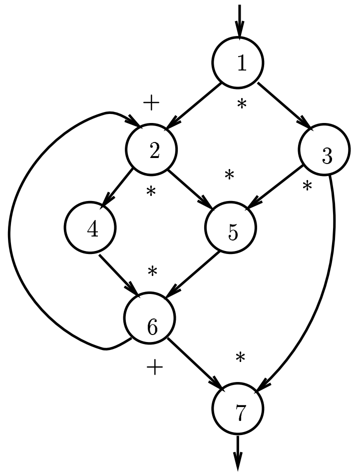
1. ${}^{*}t \in {}^{@}t$,
2. An arc x is in ${}^{@}t$ iff: (1) x is an input arc for a task which is a predecessor to t, and (2) there exists a marking at which x is marked and also t is enabled.

Figure 7 shows the enabling sets for two max nodes; the arcs that are marked by black dots are in an enabling set.

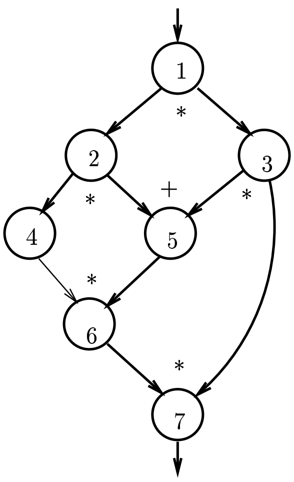
The enabling set of a max node t can be constructed using reachability analysis on some relevant portion of the task graph [5]. This construction is necessary when dealing with the



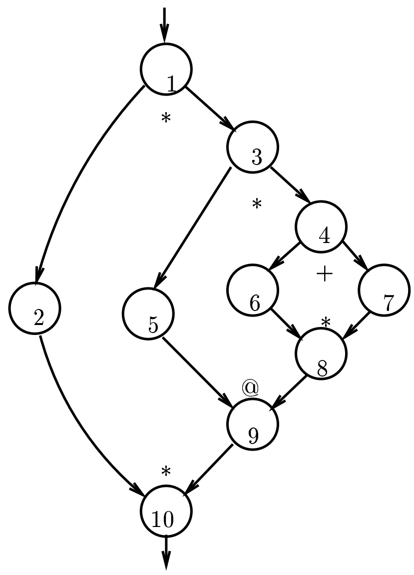
(a) neither live nor safe



(b) task graph is dead



(c) live but not safe



(d) live but t_8 is dead

Figure 6: Illegal task graphs

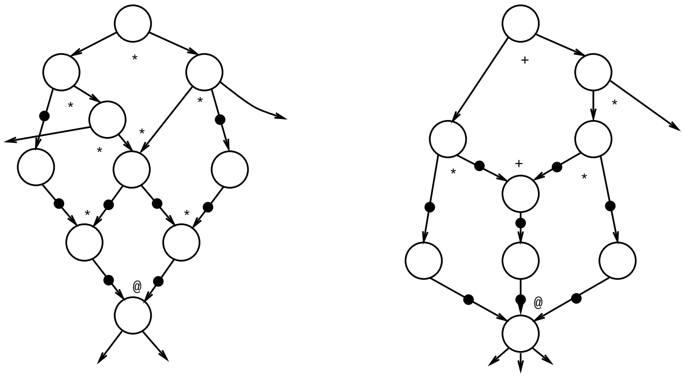


Figure 7: Enabling sets for two max nodes.

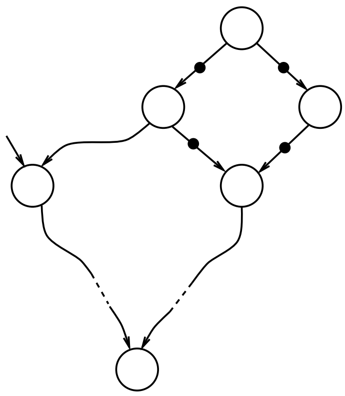


Figure 8: Illegal TG as the result of a max node.

performance aspects of the GTG's. As will be explained next, with a proper assumption, we can avoid doing so for the analysis of the GTG's.

The non-local token removal because of the firing of max nodes can create deadlocks or unsafeness in the GTG which are mainly caused by timing of the task firings. Since our analysis is based on the structural properties of the task graphs, some of these problems are hard to detect. Therefore, we shall allow the max nodes to be used only in ways that would avoid these sorts of problems. To illustrate, consider Figure 8 where t is a max node but t_b and t_c are not. It is clear that, irrespective of the type of the node t_a, the arc x belongs to the enabling set of t. However the arc y does not belong to ${}^{@}t$. In this case, we would either have a dead task or an unsafe execution. To see this, note that from the definition of the enabling sets, there exists a marking M at which x is marked and t is enabled. If t fires before t_a at M, the token in x is removed and thus y cannot get marked. If t_b is an $\frac{*}{\cdot}$ task, it cannot fire and the TG is not well-formed. Now suppose t_b is an $\frac{+}{\cdot}$ task. If z is marked at M, then by allowing t_a to fire before t and marking y, the TG can end up in an unsafe marking. If not, the same argument applies to the immediate successors of t_b and so on. If we continue this, we can find a path into which a token flows if y gets marked and remains blank otherwise. This path must join somewhere at a task t_c with the path into which the token resulting from the firing of t flows. Again, if t_c is either $\frac{*}{\cdot}$ of $\frac{+}{\cdot}$ the TG would not be well-formed. To avoid these problems, we henceforth assume the following:

Assumption 1 For a max node t, if a task $t_a \in pred(t)$ has at least one input arc in ${}^{@}t$ then all output arcs of t_a are also in ${}^{@}t$.

With this assumption, we can analyze the GTG's without the need to generate the enabling sets for the max nodes. This is done by defining a new firing rule for the max nodes as follows:

Definition 4 (new firing rule for max nodes) As before, a max node t is enabled as soon as it receives a token in one of its input arcs, say x. However, it does not fire at that time; it waits until all tokens in its enabling set that can arrive at its input arcs do so. At this point, t fires and removes all of the tokens in its input arcs (as opposed to removing from its enabling set.) It puts one token in the arc $g_t(x)$ if $OL(t) =$ '@'.

Note that this new rule is chosen only for the sake of the analysis; it does not describe the physical behavior of the max nodes. It is clear that a TG satisfying the Assumption 1 is well-formed iff it is well-formed using this new firing rule. In addition to the benefit of relieving us from the construction of the enabling sets, the new firing rule enables us to detect the dead tasks whose input arcs are in the enabling set of a max node t. With the original firing rule, the detection of such dead task is not easy since the tokens on the input arcs of such tasks are removed as a results of firing of t. This is clarified by defining a *persistent task graph* (PTG) for the TG at hand. A PTG models the repeated execution of the TG.

Definition 5 (persistent task graph (PTG)) A persistent TG is constructed from a TG by connecting its initial and terminal arcs. In a PTG, the initial task is enabled immediately after the terminal task fires. The firing rule for the max nodes in a PTG is in accordance to the Definition 4. A PTG is live iff its TG is live.

The main purpose for above definitions is the following lemma based on which we can break up the well-formedness problem of a GTG into two separate problems, namely its liveness and its safeness problems.

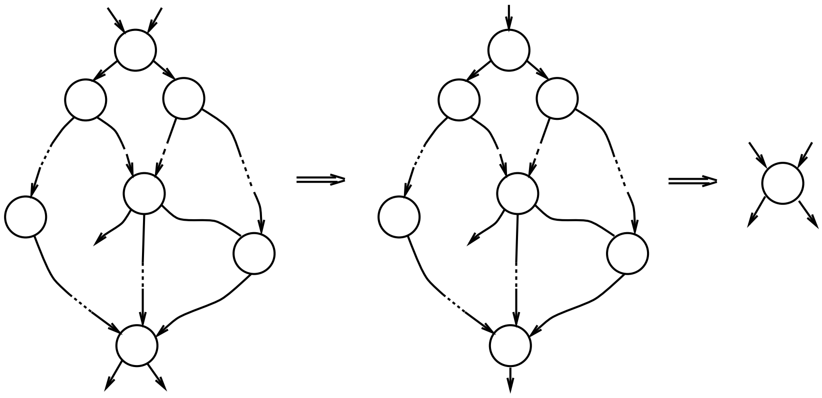


Figure 9: Elimination of $\frac{@}{@}$ nodes when possible.

Lemma 1 *A GTG is well-formed iff its PTG is live and safe.*

Proof: The only if part is obviously true. If the GTG is not well-formed, it is either not live or unsafe, in which case the PTG is also dead or unsafe, or there exists some tasks that cannot never fire. For the latter case, there must exist a $\frac{*}{\cdot}$ task t with an input arc x where a token arrives but gets stuck. In the GTG, if x is in a enabling set of a max node the token at x is removed, but this is not so in the PTG, because of the new firing rule for max nodes. Therefore, if the PTG is not dead, x can get unbounded number of tokens. $\square$

4 Analysis of the Generalized Task Graphs

In this section we present the main results regarding the well-formedness of the generalized task graphs. As explained in the introduction, a straight forward approach to this problem is the one which examines all possible reachable markings of a task graph and checks that they do not violate the conditions given in the Definition 2. For the cyclic task graphs, this method can be extremely expensive because, the number of the reachable markings can be quite large.

The other possible approach to the problem is based on the static structural properties of the GTG's which we discuss in the rest of the paper. We will provide the necessary and sufficient conditions for the well-formedness of a GTG using this approach.he The results that we present here are only applicable to the GTG's with only $\frac{@}{+}$ and/or $\frac{@}{*}$ nodes. However, the method is still good for the GTG's which uses $\frac{@}{@}$ nodes in a somewhat restricted form. The restriction is illustrated in the Figure 9. If the output arcs of the $\frac{@}{@}$ node t are followed in all forward directions, we can create a sub-task graph Σ_t with t as its "initial" task and another task t' as its "terminal" task. If the input arcs of the tasks in Σ_t have all been visited in constructing Σ_t, i.e., if no arcs from other parts of the task graph to the tasks in Σ_t exist, then we can reduce the problem to the one for another similar GTG without $\frac{@}{@}$ nodes. This can easily be done by first applying the algorithm to the sub-task graph Σ_t. If it is well-formed, we can then "compress" Σ_t and replace the $\frac{@}{@}$ node t with one such that $OL(t) = OL(t')$.

Regarding the cycles in the TG's, there are some easily detectable problems. Recall that a cycle C is uniquely identified by a backward arc x and two non-simple tasks t_i and t_j that are connected via x. Task t_j must be of type $\frac{\cdot}{+}$ otherwise it creates an infinite loop and an unbounded number of tokens in the TG. Similarly, if t_i is a $\frac{*}{\cdot}$ task, it is dead. Therefore, for the rest of this paper, we assume that the TG does not contain cycles with these problems. Summary of assumptions: It is assumed that the GTG subject to our analysis satisfy the following properties:

1. It does not contain $\frac{@}{@}$ nodes.
2. The max nodes satisfy the Assumption 1.
3. The head task of a cycle in the GTG is not a $\frac{*}{\cdot}$ task and its tail task is a $\frac{\cdot}{+}$ task (no infinite loops.)

We now present the main results of this paper. We follow an approach similar to that of Hack's [6]. Using Definition 5 and Lemma 1, we consider the liveness problem of a GTG first. Next, assuming that the GTG is live, we deal with its safeness.

We start out with the definition of traps and deadlock-sets which are sets of arcs in a GTG and provide their properties which lead to the final results.

4.1 Traps and Deadlock-Sets

Similar to the traps and deadlocks defined for Petri nets [6], we define these subsets of arcs as follows (we use the term 'deadlock-set' for what is normally called a 'deadlock' in Petri nets to avoid confusing it with the deadlock situation that occurs in the system.) A trap is a set of arcs that remains marked if it ever receives a token. Similarly, a deadlock-set is a set of arcs that remains blank forever if it gets blank. In the following, we give a formal definition of the traps and deadlock-sets which are methods for their constructions.

Definition 6 (trap) A set of arc $T \subseteq E$ in a TG is a trap if it satisfies the following properties:

1. The initial arc a_s is in T.
2. If arc $x \in T$ and $x \in {}^*t$ for an $\frac{\cdot}{*}$ task t, then at least one of the output arcs of t must also be in T.
3. If arc $x \in T$ and $x \in {}^*t$ for an $\frac{\cdot}{+}$ or a max node t, then all of the output arcs of t must also be in T.

Definition 7 (deadlock-set) A set of arcs $D \subseteq E$ in a TG is a deadlock-set if it satisfies the following properties:

1. The terminal arc is in D.
2. If arc $x \in D$ and $x \in t^*$ for an $\frac{*}{\cdot}$ task t, then at least one of the input arcs of t must also be in T.
3. If arc $x \in D$ and $x \in t^*$ for an $\frac{+}{\cdot}$ or a max max t, then all of the input arcs of t must also be in T.

A trap (deadlock-set) is called *minimal* if no subset of it is a trap (deadlock-set.) The following properties can be shown easily:

1. The set of all arcs in a GTG is the maximal trap and also the maximal deadlock-set.
2. A trap is always marked.
3. If a deadlock-set is blank, it remains blank forever.
4. Because of the assumptions we made for the cycles in the TG (Assumption 3,) the subsets of arcs that satisfy the properties (2) and (3) above must contain the initial and terminal arcs.
5. If T_1 and T_2 (D_1 and D_2) are two different traps (deadlock-sets) then $T_1 \cup T_2$ ($D_1 \cup D_2$) is also a trap (deadlock-set.)

Given a set of arcs $B \subseteq E$, we can define a *sub-task graph* over B such that it includes only those nodes which have at least one input or output arc included in B. Formally,

Definition 8 (sub-task graph) A sub-task graph Σ_B defined over a set of arcs B is a task graph $(W_B, \prec_B, IL_B, OL_B)$ such that $W_B = {}^*B \cup B^*$ is the set of tasks in Σ_B and $\prec_B = \prec \cap B$ is the precedence relations on W_B. The input and output labels of a tasks $t \in W_B$, namely $IL_B(t)$ and $OL_B(t)$, are the same as $IL(t)$ and $OL(t)$ respectively.

A sub-task graph defined over an arbitrary set of arcs may not represent a meaningful task graph. But, here we are only interested in sub-task graphs defined over the traps and deadlock-sets. Such sub-task graphs have all the properties of complete task graphs and can be treated as such. Also note that for a trap/deadlock-set B we have $W_B = {}^*B = B^*$. Figure 10 shows a GTG, which happens to be live but not safe, with its traps and deadlock-sets and two sub-task graphs.

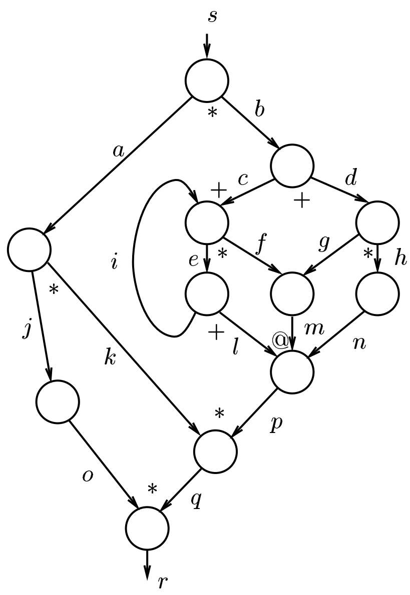
In this paragraph, we intuitively explain how a trap or a deadlock-set can be constructed. To construct a trap, we start from the initial arc a_s and move along the arcs of the TG. When we reach a task, we follow along its output arc(s) to satisfy the properties of Definition 6. This is continued along all paths until we reach the terminal arc a_e. A deadlock-set is found similarly: We start from a_e but move in the opposite directions of the arcs in the TG. When a task is reached, we follow toward its input arc(s) according to the Definition 7. This is continued until we reach the initial arc. All arcs on the paths we follow are in a the trap/deadlock-set. By choosing only one of the output (input) arcs of $\frac{\cdot}{*}$ ($\frac{*}{\cdot}$) tasks on the path we would construct a minimal trap (deadlock-set.)

4.2 The Liveness Problem

In this section, we present the necessary and sufficient conditions for the liveness of a GTG. As we defined before, a GTG is live iff the terminal task fires at the end of every sequence of firings. First, we provide the sufficient condition.

Lemma 2 (liveness sufficient condition) *If at every marking reachable from M_0, the deadlock-sets of a GTG are all marked, then the GTG is live.*

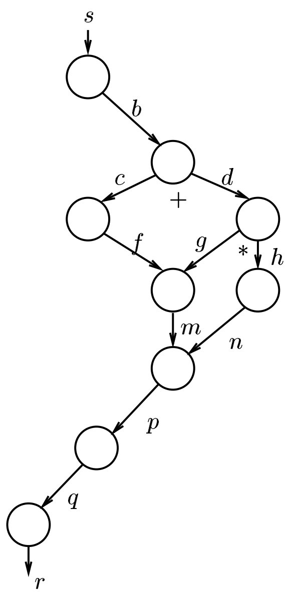
Proof: We will prove the opposite, that is, if the GTG is dead, then there must exist some marking at which at least one deadlock-set is blank. Suppose that the GTG is dead, then, according to the definition, there exists a marking M at which no sequence of firing can enable



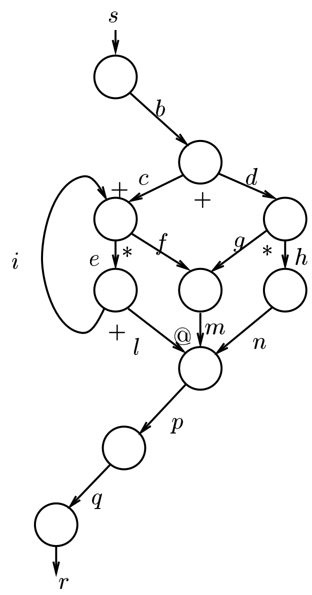
minimal traps:
$\{s,a,j,o,r\}$
$\{s,a,k,q,r\}$
$\{s,b,c,d,f,h,m,n,p,q,r\}$
$\{s,b,c,d,f,g,m,p,q,r\}$
$\{s,b,c,d,e,h,i,l,n,p,q,r\}$
$\{s,b,c,d,e,g,i,l,m,p,q,r\}$

minimal deadlock-sets:
$\{r,o,j,a,s\}$
$\{r,q,k,a,s\}$
$\{r,q,p,l,m,n,k,e,f,g,h,c,d,b,s\}$

(a)



(b) the sub-task graph defined over a trap



(c) the sub-task graph defined over a deadlock-set

Figure 10: Traps and deadlock-sets.

the terminal task. This means that all the firing sequences from M will put the TG either in some infinite loops or in a dead marking at which no task can fire. The former is impossible because we assumed that there is no cycle in the task graph that creates infinite loop. Let M_d be one such dead marking. At M_d, the input arcs of $\frac{+}{\cdot}$ and max tasks are all blank and for all tasks t of type $\frac{*}{\cdot}$ there exists at least one arc $a \in {}^*t$ which is blank. Now, based on the informal method given in the last section, we can construct a blank minimal deadlock-set. $\square$

For the proof of the next important theorem, we need the conditions under which a deadlock-set does not contain any trap. We will provide these conditions in the next lemma, but to do so, we need some further notation.

Given a trap T and a task $t \in T^*$ (t has at least one input/output arc in T,) let the *partial trap* T_t denote the subset of T which is "accessible" from t, i.e., if we follow the directed arcs from t, then the arcs in T that are visited constitute T_t. Given a partial trap T_t, we can always construct a complete trap T which may not be unique.

Now, given a deadlock-set D and a task $t \in D^*$ we use the expression $T_t \not\subseteq D$ as a shorthand notation to indicate that it is impossible to construct a partial trap T_t from task t that is contained in D. Formally, $T_t \not\subseteq D$ means that $\nexists$ a trap $T : T_t \subseteq D$, and $T_t \subseteq D$ implies the opposite.

Lemma 3 *Let D be a minimal deadlock-set and $t \in D^*$. Then,*

1. *If t is of type $\frac{\cdot}{*}$, a partial trap $T_t \subseteq D$ exists iff for at least one immediate successor t_i of t there exists a partial trap $T_{t_i} \subseteq D$. Formally, $T_t \subseteq D \Longleftrightarrow \exists t_i \in isucc(t) : T_{t_i} \subseteq D$.*
2. *If t is of type $\frac{\cdot}{+}$, then a partial trap $T_t \subseteq D$ exist iff,,*
   - *(a) All the output arcs of t is in D, and*
   - *(b) For every $t_i \in isucc(t)$ there exists a partial trap $T_{t_i} \subseteq D$.*

Proof: Follows trivially from the definitions of deadlocks and traps. $\square$

The necessary condition for the liveness of a GTG can now be stated as follows.

Theorem 1 (liveness necessary condition) *If there exists a minimal deadlock-set D which does not contain any traps, then there exists a sequence of firing after which the GTG becomes deadlocked.*

Proof: Starting at the initial marking M_0, we will provide a sequence of firings that puts the GTG in a marking from which no sequence of task firing can enable the terminal task. It will be shown that the following procedure creates such firing sequences:

(1) Fire all enabled tasks many times as possible.

(2) For a $\frac{\cdot}{+}$ task t, do the following for the choice of the output arc that receives tokens:

(2.1) if $t \in D^*$ then it will be shown that there must exist an output arc $x \in t^*$ such that either $x \notin D$ or if t_i is the output task of x ($t_i \in isucc(t)$) then $T_{t_i} \not\subseteq D$. Put one token in x for each firing of t.

(2.2) if $t \notin D^*$ the choice of the output arc is arbitrary.

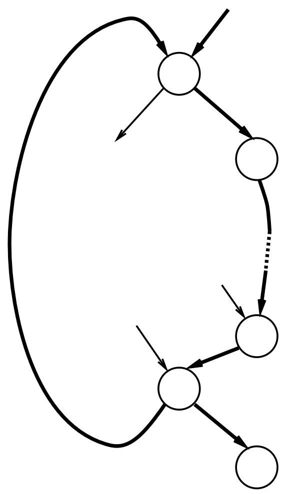


Figure 11: A general cycle in the task graph.

We claim that if we follow this procedure, a task $t \in D^*$ that is enabled must satisfy the condition $T_t \nsubseteq D$. In other words, if $T_t \subseteq D$ then task t cannot be enabled. If this claim is true, so is the claim in the step 2.1 of the firing procedure. For the proof, let $M_0 \rightarrow M_1 \rightarrow \ldots$ be the subsequent markings that are resulted from this firing procedure. We shall use induction on these markings to prove the claim. The hypothesis, i.e., $T_{t_1} \nsubseteq D$ where t_1 is the initial task, is the basis for the induction. Now, assume that the claim is true for all M_i, $i < k$. Let $t \in D^*$ be the task that is enabled at the marking M_k. For all possible types of t, i.e., $\frac{*}{\cdot}$, $\frac{+}{\cdot}$, or $\frac{@}{\cdot}$, there must exist an immediate predecessor for t, say $t_a \in D^*$ whose firing at a marking M_i, $i < k$, has caused the enablement of t. By the induction hypothesis we have $T_{t_a} \nsubseteq D$. Now, if t_a is a $\frac{\cdot}{*}$ task, Lemma 3 implies that $T_t \nsubseteq D$. If, on the other hand, t_a is a $\frac{\cdot}{+}$ task, according to the same lemma again, either: (1) there exists an output arc for t_a, say x, that is not in D, or (2) there exists an immediate successor for t_a, say t_b, such that $T_{t_b} \nsubseteq D$. The first possibility is a contradiction to the assumption that t has been enabled right after t_a because, in that case, according to the firing procedure, the token from the firing of t_a would have been forwarded to arc x which is not the input arc of t. In the second case, using a similar argument, t_b cannot be any other task but t. Therefore, $T_t \nsubseteq D$.

Note that, the firings sequence σ that is resulted from the above firing procedure is not unique. However, no such σ can fire the terminal task t_e because the partial trap T_{t_e} contains only the terminal arc which is also in D. Therefore, σ will eventually put the task graph either in a marking at which no task can fire or in an infinite cycle. We will prove that second case is not possible.

Suppose that the task graph ends up in some infinite cycle. Firstly, such marking cycle must have been caused by the enforcement of the firing procedure, i.e., the choices for the output token of the $\frac{\cdot}{+}$ tasks in these cycles should not have been made arbitrarily. Because otherwise, it is not possible to have an infinite cycle of markings in a TG that, by assumption, does not contain infinite loops. Secondly, there must exist a cycle in the TG corresponding to each marking cycle. Let C be such a cycle in the TG. C contains a backward arc y and two non-simple tasks t_1 and t_2 that are connected via y. Figure 11 shows such a typical cycle where the thicker arcs are assumed to be in D. By assumption, t_1 is not a $\frac{*}{\cdot}$ task and t_2 is a $\frac{\cdot}{+}$ task. Having a marking cycle corresponding to C implies that:

1. $y \in D$ and t_1 and t_2 are both in D.

2. $T_{t_1} \not\subseteq D$ and $T_{t_2} \not\subseteq D$, otherwise t_1 and t_2 could never fire in σ.

3. $t_2{*} \in D$ and for all $t_3 \neq t_1$ that is an immediate successor of t_2, we must have $T_{t_3} \subseteq D$. Otherwise, the firing procedure also fires such a $t_3 \neq t_1$ which results in breaking the cycle.

We prove that task t_2 satisfying above properties can never fire in σ in the first place. This is accomplished by noticing that if t_2 is firable in σ, it must also be firable if we remove the backward arc y from the TG. But this is impossible because, by removing y we have $T_{t_2} \subseteq D$ which implies that t_2 can never fire in σ.

This completes the proof for the claim that, following the firing procedure, it is not possible to put the TG in an infinite cycle which implies that the TG must reach a marking at which no task is enabled. $\square$.

We conclude the following theorem which is exactly the same as that for the liveness of free choice Petri nets. As we stated before, the GTG's are more powerful than free choice Petri nets and thus this result cannot be concluded directly from the one in [6] by Hack.

Theorem 2 (liveness) *A GTG is live iff every deadlock-set contains a trap.*

Corollary 1 *A GTG whose tasks are all either of types* $\frac{*}{*}$ *and* $\frac{@}{*}$ *or of types* $\frac{+}{+}$ *and* $\frac{@}{+}$ *is live.*

Proof: If all tasks are of types $\frac{*}{*}$ and $\frac{@}{*}$ then any path from the initial to the terminal task is a trap. Thus, any deadlock-set contains at least one trap. This is also true if the tasks are all of types $\frac{+}{+}$ and $\frac{@}{+}$ In this case the set of arcs E is the only deadlock-set and trap in the task graph. $\square$

4.3 The Safeness Problem

In this section, we assume that the TG is live, that is, it always terminates. Therefore a deadlock-sets contain a trap and thus is always marked. With this assumption, we will derive the necessary and sufficient conditions for the safeness of a persistent TG (PTG). As was shown in Lemma 1, a GTG is well-formed iff its PTG is live and safe. If the PTG is not safe, either the original TG is not safe, or there exists a task in the TG that cannot fire.

The safeness of a PTG is characterized by studying the properties of the sub-task graph defined over the deadlock-sets. As stated earlier, a sub-task graph defined over a deadlock-set and a trap has all the properties of an independent TG and can be viewed as such. As a result, a persistent sub-task graph (PSTG) is a meaningful task graph. Note that the sub-task graph Σ_D defined over any minimal deadlock-set D is live even if the original TG was dead because such a Σ_D does not contain any $\frac{*}{\cdot}$ tasks which are the only nodes that cause a deadlock. With these in mind, we present the main theorem for the safeness of a PTG.

Theorem 3 *A live PTG* Σ *is safe iff the persistent sub-task graphs defined over its minimal deadlock-sets are all safe.*

Proof: For an arbitrary minimal deadlock-set D, let Σ_D be the PSTG defined over D. For the "if" part , assume that all Σ_D's are safe. We construct a new sub-task graph Σ'_D which is the same as Σ_D but it also includes the output arcs of the $\frac{\cdot}{+}$ tasks $t \in D^*$ that are not in D. Let R_D denote the set of these additional arcs. There is no output task for an arc $x \in R_D$ in Σ'_D and if x gets marked at a marking of Σ'_D, the token is simply removed from Σ'_D in the next marking. Clearly, Σ_D is safe iff Σ'_D is safe. Let Q and Q_D be the sets of all the reachable markings of Σ and Σ'_D respectively. Also for a marking $M \in Q$ let the partial marking $Z_D(M) \subseteq M$ represent the markings (number of tokens) of the arcs in Σ'_D (i.e., $D \cup R_D$) at M.

We claim that for any marking $M \in Q$, there exists a marking $M_D \in Q_D$ for Σ'_D such that $Z_D(M) = M_D$. The correctness of this claim would imply that, if Σ'_D is safe for every D, then Σ must also be safe, since any $M \in \vec{M}$ is the union of the partial markings $Z_D(M)$ over all minimal deadlocks D's. We use induction on Q to prove this claim. The claim is trivially true for the initial marking M_0. Suppose that it is also true for a reachable marking M. That is, there exists a marking $M_D \in Q_D$ such that $Z_D(M) = M_D$. Let t be the task that is enabled at M and let M' be the resulting marking after the firing of t, i.e., $M \xrightarrow{t} M'$. Such a task exists because the TG is live. If t is not in Σ'_D, then, from the properties of the deadlock-sets, the firing of t cannot inject a token in Σ'_D neither it can remove tokens from D. It may remove tokens from an arc $x \in R_D$, however, but such a removal occurs in Σ'_D at the marking M_D also. Therefore, M' also has the claimed property. If, on the other hand, $t \in D^*$, then t must also be enabled in Σ'_D at M_D because there is no $\frac{*}{\cdot}$ tasks in Σ'_D. If $M_D \xrightarrow{t} M'_D$, then clearly, no matter which type a task t is ($\frac{\cdot}{*}$ or $\frac{\cdot}{+}$), the resulting markings would satisfy $Z_D(M') = M'_D$.

For the converse, assume that Σ is live and safe but there exists an unsafe sub-task graph Σ_D. Since there is no $\frac{*}{\cdot}$ task in Σ_D, there must exist a $\frac{\cdot}{*}$ task t_1 and a $\frac{+}{\cdot}$ task t_2 in Σ_D such that the tokens that are created after the firing of t_1 are eventually merged in t_2. These tasks and the arcs that carry the tokens from t_1 to t_2 are also in Σ. Also, the liveness and safeness of Σ implies that any token that arrives at an input arc of a task t should eventually cause the firing of t. If t is a $\frac{\cdot}{+}$ task, after its firing, we are free to choose any output arc of t to receive a token. Based on this observation, it is easy to see that, after t_1 fires in Σ, we can force the resulting tokens to follow the same arcs as they do in Σ_D and arrive at t_2. This is a clear contradiction to the assumption that Σ is safe which completes the proof. □

The necessary and sufficient conditions for the safeness of a PSTG Σ_D defined over a deadlock-set D was implicitly mentioned in the last theorem. Formally,

Lemma 4 *A PSTG Σ_D defined over a deadlock-set D is* unsafe *iff, in the graph representation of Σ_D, there exists a $\frac{\cdot}{*}$ node t_1 and a $\frac{+}{\cdot}$ node $t_2 \in succ(t_1)$ that are in the same biconnected components.*

Proof: Suppose there exists such a biconnected component, i.e., there exists at least two paths from t_1 to t_2 that do not meet at any other nodes. Since Σ_D is live for any D, these two paths can carry the tokens created from the firing of t_1 to t_2 and create unsafeness. If no such component exists, Σ_D is clearly safe. □

Corollary 2 *A PSTG Σ_D defined over a minimal deadlock-set D that does not contain any $\frac{@}{\cdot}$ node is safe iff it does not include any $\frac{\cdot}{*}$ tasks.*

Corollary 3 *A GTG whose tasks are all either of types $\frac{*}{*}$ and $\frac{@}{*}$ or of types $\frac{+}{+}$ and $\frac{@}{+}$ is well-formed.*

Proof: From Corollary 1, such a GTG is live. Now, if the tasks are all of types $\frac{*}{*}$ and $\frac{@}{*}$ then because of the Lemma 4 all PSTG's defined over the deadlock-sets are safe since there are no $\frac{+}{\cdot}$ tasks. In the case where the task types in the GTG are all of types $\frac{+}{+}$ and $\frac{@}{+}$, the number of tokens is always one and thus the GTG is safe. □

And finally, we conclude with the following theorem which can also be deduced directly from the liveness-safeness results for the free-choice Petri net [6].

Theorem 4 *An extended task graph is well-formed iff every minimal deadlock-set is also a minimal trap.*

Proof: Assume that the TG is well-formed . Then from Theorem 2 every minimal deadlock-set D must contain at least one trap. Also, since its PTG is safe, from Corollary 2, there is no $\frac{\cdot}{*}$ task in the sub-task graph Σ_D defined over a deadlock-set D. In other words, Σ_D contains only $\frac{+}{+}$ and simple tasks. But, this is also the characterization of a sub-task graph defined over a minimal trap. Therefore, a deadlock-set D must also be a trap.

Conversely, assume that every minimal deadlock-set D is also a minimal trap. From the liveness theorem, the TG is live. The sub-task graph defined over D should not have any $\frac{\cdot}{*}$ task since D is also a minimal trap. Therefore according to Corollary 2 again, the PTG must also be safe which completes the proof. □

5 Conclusions

In this paper, we investigated the properties of the generalized task graphs which are the extended task graphs (i.e., fork-join, branch-merge task graphs) augmented with a new node type called *max* node. The generalized task graphs can be used to model a richer class of parallel computations including the non-deterministic parallel programs. In non-deterministic algorithms, often, the verification of all possible solutions to a problem is carried out concurrently and the first guess that is verified is the solution to the whole problem. Such task graphs are prone to problems such as deadlock, unboundedness, and unsafeness. Therefore, it is necessary to verify that a given task graph is free of these problems. We defined the notion of well-formedness of task graphs by viewing them as high level Petri nets and provided necessary and sufficient conditions under which a generalized task graph is well-formed. These conditions are based on the structural properties of the task graph and hence is, in general, less expensive compared to those based on the standard reachability analysis. However, these methods suffer from the exponential complexity in the worst case, because the deadlock detection in such a system is an NP-complete problem.

References

[1] J. L. Baer, *Graph models of computations in computer systems*, PhD thesis, Dept. of Elect. Engg., UCLA, Los Angeles, CA 90024-1600, 1968.

[2] J. L. Baer, D. P. Bovet, and G. Estrin, *Legality and other properties of graph models of computations*, Journal of ACM, 17 (July 1970), pp. 543–554.

[3] F. Commoner and A. W. Holt, *Marked directed graphs*, Journal of Computer and System Science, (1971), pp. 511–523.

[4] J. B. Dennis, *Data flow supercomputers*, IEEE Computer, (Nov. 1980), pp. 48–56.

[5] M. Ghodsi and K. Kant, *Properties of the generalized task graphs as models for parallel computations*, Tech. Report CS-88-50, Computer Science Dept., Penn State University, Univ Park, PA 16801, Dec. 1988.

[6] M. Hack, *Analysis of production schematic by Petri nets*, master's thesis, Lab. of Computer Science, MIT, Cambridge, MA, 1972. (tech report# 94, Project MAC).

[7] N. D. Jones, L. H. Landweber, and Y. E. Lien, *Complexity of some problems in Petri nets*, Theoretical Computer Science A, (1977), pp. 277–299.

[8] K. Kant, *Analysis and synthesis of generalized task graphs*, in Annual Allerton Conference, Urbana, IL, 1989.

[9] R. M. Karp and R. E. Miller, *Properties of a model for parallel computations: determinacy, termination, queueing*, SIAM J. Applied Math, 14 (Nov. 1966), pp. 1390–1411.

[10] K. Kavi, B. Buckles, and U. N. Bhat, *A formal definition of data flow models*, IEEE Transactions on Computer, C-35 (Nov. 1986), pp. 940–948.

[11] ——, *Isomorphism between Petri nets and dataflow graphs*, IEEE Transactions on Software Engineering, SE-13 (Oct. 1987), pp. 1127–1134.

[12] R. E. Miller, *A comparison of some theoretical models of parallel computation*, in Proc. Symp. Automat. Computation and Contr ., Milwaukee, WI, Apr. 1975, pp. 337–343.

[13] J. L. Peterson, *Petri Net Theory and The Modeling of Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[14] W. Reisig, *Petri Nets: An Introduction*, Springer-Verlag, 1982.

[15] R. A. Sahner and K. Trivedi, *Performance and reliability analysis using directed acyclic graphs*, IEEE Transactions on Software Engineering, SE-13 (Oct. 1987), pp. 1105–1114.