A Structural Approach to Quasi-Static Schedulability Analysis of Communicating Concurrent Programs*

Cong Liu
Department of EECS
UC, Berkeley
Berkeley, CA 94720
congliu@eecs.berkeley.edu

Alex Kondratyev, Yosinori Watanabe
Cadence Berkeley Labs
Berkeley, CA 94704
{kalex, watanabe}@cadence.com

Alberto Sangiovanni-Vincentelli
Department of EECS
UC, Berkeley
Berkeley, CA 94720
alberto@eecs.berkeley.edu

ABSTRACT

We describe a system as a set of communicating concurrent programs. Quasi-static scheduling compiles the concurrent programs into a sequential one. It uses a Petri net as an intermediate model of the system. However, Petri nets generated from many interesting applications are not schedulable. In this paper, we show the underlying mechanism which causes unschedulability in terms of the structure of a Petri net. We introduce a Petri net structural property and prove unschedulability if the property holds. We propose a linear programming based algorithm to check the property, and prove the algorithm is valid. Our approach prove unschedulability typically within a second for Petri nets generated from industrial JPEG and MPEG codecs, while the scheduler fails to terminate within 24 hours.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application Based Systems**]: Real-time and embedded systems; D.2.2 [**Software Engineering**]: Design Tools and Techniques —*Petri nets*

General Terms

Algorithms, Design, Theory

Keywords

Quasi-static scheduling, Petri nets, structural analysis

1. INTRODUCTION

The complexity of embedded systems has been increasing dramatically. It forces designers to adopt formal models to describe system behaviors and hide implementation details. It is often necessary to decompose a complicated system into functional entities so that the design complexity of each entity is manageable. Concurrent models, such as dataflow networks [14], Kahn process networks (KPN) [12][13], and Communicating Sequential Processes [11], are suitable for these purposes. In a concurrent model, a system consists of a set of processes. Each is described by a sequential program. The processes communicate with the environment and other processes, and run at their own speed.

However, the concurrent processes often share a physical resource, e.g. a CPU or a bus. Hence, their implementation often requires solving a fundamental scheduling problem, i.e. sequencing the operations of concurrent processes under certain constraints. Depending on how and when the scheduling decision are made, scheduling algorithms could be classified as: static, quasi-static, and dynamic. Dynamic scheduling makes all scheduling decisions at run-time. It introduces context switch overhead. Static scheduling makes all scheduling decisions at compile-time. It reduces the context switch overhead, because no run-time scheduling decision has to be made. Due to this static scheduling is restricted to systems without data-dependent choices, e.g. *if-then-else*. Quasi-static scheduling [7] is applied to systems in which data-dependent choices occur. It performs static scheduling as much as possible while leaving data-dependent control to be resolved at run-time.

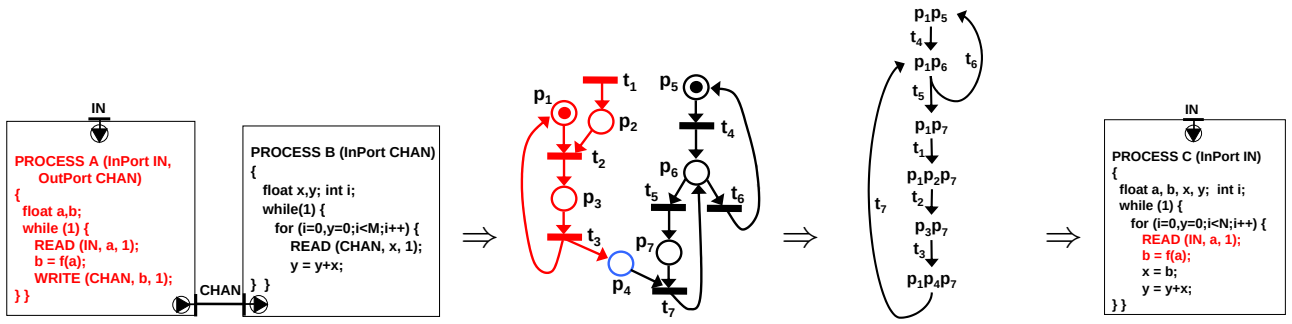
The existence of a quasi-static schedule is proved to be undecidable for Boolean dataflow networks [6]. The most recent advancement [7] on quasi-static scheduling uses a Petri net [16] as an intermediate representation of the system specification. Its scheduling flow is shown in Figure 1. First, the concurrent programs are transformed into a Petri net where program statements are represented by transitions, and system states correspond to PN markings. Then a heuristic algorithm searches the reachability tree of the Petri net for a schedule. If it succeeds, the generated schedule is transformed back to a sequential program. Note that during the transformation, data-dependent choices are abstracted as non-deterministic free choices. Consequently, a scheduler considers all possible choice outcomes at any state. The abstraction helps to provide an efficient method to find a schedule. However, the conservative consideration makes many interesting applications unschedulable.

1.1 Motivation and Contribution

The quasi-static scheduler searches for a schedule in a typically infinite space bounded by some heuristic. Thus, if a Petri net is unschedulable, the scheduler has to com-

*This research was partly supported by MARCO Gigascale Systems Research Center award 2003-DT-660.

EMSOFT'05, September 19–22, 2005, Jersey City, New Jersey, USA.



(a) System specification as communicating concurrent programs.

(b) Petri net obtained from the system specification.

(c) A quasi-static schedule of the Petri net.

(d) A quasi-static schedule of the system specification.

Figure 1: Quasi-static scheduling flow.

plete the exhaustive search before reporting unschedulability. Furthermore, heuristic-based schedulers can not prove unschedulability even if they fail to find a schedule. Since many interesting applications are unschedulable, schedulability analysis before constructing a schedule is desirable.

It is common that concurrent processes representing an application exchange messages while repeating the same computation pattern in a loop. Processes participating in an exchange might have correlated conditions for exiting their loops. For example, two *for* loops might have the same number of iterations. Therefore it is not needed to consider all possible true/false combinations of the two conditions. Arrigoni [4] proposed an approach to modify the original programs so that one of the two control structures having the same exit condition would not be modeled by a free choice. It requires designers to manually label the correlated conditions in programs, and thus is not practical. Standard static program analysis techniques, such as abstract interpretation [8], could be applicable. But the technique is known to be computationally expensive, and can only deal with arithmetic operations on variables of a system.

Our approach is different from the known ones. We explore the implication of the correlated conditions in the original programs to their structures in the generated Petri net. That is, we study the structure of the generated Petri net and identify a structural property that causes unschedulability. By exposing the structural correlation, we get hints on the correlated conditions in original programs.

We observe that regular and repeated structural patterns exist in schedules of Petri nets generated from many applications. These patterns result in infinite paths that prevent a schedule to return to a designated set of states (initial state in particular), which causes unschedulability. The phenomenon is driven by the definition of a schedule and, interestingly enough, a structural property of the Petri net. Based on this structural property, we formulate and prove a sufficient condition to check unschedulability. The condition does not involve markings of a Petri net, and can be applied to general Petri nets. We propose an algorithm based on linear programming to efficiently check the condition. Once unschedulability is determined, the algorithms provides hints on the location of correlated data-dependent conditions in programs. This facilitates further handling and resolving unschedulability. Our approach is shown to be effective and efficient based on its application to Petri nets generated from industrial JPEG MPEG codecs and comparing performance with a heuristic-based scheduler.

1.2 Paper Organization

The rest of the paper is organized as the following. Section 2 reviews the basic definitions and notations of Petri nets. In Section 3, we define a Petri net structural property and prove a sufficient condition for unschedulability. Section 4 presents a linear programming based algorithm to check the property, and prove the validness of the algorithm. Section 5 illustrate the advantages of our approach using JPEG and MPEG codecs as the driver applications. Finally, Section 6 discusses the limitations our approach.

2. PRELIMINARIES

A *Petri net* is a 4-tuple (P, T, F, M_0). $P = \{p_1, p_2, \ldots, p_m\}$ is a set of *places*. $T = \{t_1, t_2, \ldots, t_n\}$ is a set of *transitions*. $F\colon (P \times T) \cup (T \times P) \to \mathbb{N}$ is the flow relation. $M_0\colon P \to \mathbb{N}$ is the *initial marking*. In general, $M : P \to \mathbb{N}$ is a *marking*, which represents a state of a Petri net. $\mathbb{N}$ denotes the set of nonnegative integers. Let $N = (P, T, F)$ denote a Petri net structure without any specific initial marking, and (N, M_0) denote a Petri net with the given initial marking.

A Petri net can be represented by a directed, weighted, bipartite graph as shown in Figure 1(b), in which circles denote places, bars denote transitions, and directed arcs denote the flow relation. A marking is represented by an assignment of black dots in places.

A transition t is *enabled* at a given marking M, if $M(p) \geq F(p, t)$ for all place $p \in P$. When a transition is enabled it can *fire*. The new marking M' reached after the firing of t is defined as: $M'(p) = M(p) - F(p, t) + F(t, p)$ for all place $p \in P$.

A marking M is *reachable* from the initial marking M_0 if there exists a sequence of firings that transforms M_0 to M. It is denoted by $M_0[\sigma > M$, where σ represents a *firing sequence* $(t_{\sigma 1}, t_{\sigma 2}, \cdots, t_{\sigma k})$. The *firing count vector* $\bar{\sigma}$ of a firing sequence σ is a $|T|$-vector, where the ith entry denotes the number of times that transition t_i fires in σ. The set of reachable markings from the initial marking is denoted by $R(N, M_0)$.

In this paper we use nets with *source* transitions, i.e. with empty pre-sets. These transitions model the behavior of the input stimuli to a reactive system. Their set is denoted by T_S.

The incidence matrix $\boldsymbol{A} = [a_{ij}]$ is a $|T| \times |P|$ matrix, where $a_{ij} = F(t_i, p_j) - F(p_j, t_i)$. A vector $\boldsymbol{x} \in \mathbb{N}^{|T|}$ is called a *T-invariant* if $A^{\mathrm{T}}\boldsymbol{x} = 0$. A T-invariant $\boldsymbol{x}$ is said to be *minimal* if there exists no T-invariant $\boldsymbol{x}' \neq 0$ with $\boldsymbol{x}' \leq \boldsymbol{x}$. It is known that a $|T|$-vector $\boldsymbol{x}$ is a T-invariant if and only if there exists a marking M and a firing sequence σ from M back to M with $\bar{\sigma} = \boldsymbol{x}$. For example, $(1\ 1\ 1\ 0\ 1\ 0\ 1)^{\mathrm{T}}, (0\ 0\ 0\ 1\ 0\ 1\ 0)^{\mathrm{T}}$ are T-invariants of the Petri net in Figure 1(b) and they are also minimal T-invariants. Let X denote the set of all T-invariants of a Petri net. The set of transitions corresponding to non-zero entries in a T-invariant x is called the *support* of an invariant and is denoted by $\|\boldsymbol{x}\|$. We say t is contained in $\boldsymbol{x}$ or $\boldsymbol{x}$ contains t if $t \in \|\boldsymbol{x}\|$.

A *free choice set* (FCS) is a maximal subset of transitions S such that $\forall t_1, t_2 \in S$, $F(p, t_1) = F(p, t_2)$ for all $p \in P$. FCSs have the property that there exists no marking for which one transition in an FCS is enabled while another is disabled. In other words, transitions in an FCS are enabled at the same time. For example, $\{t_5, t_6\}$ is the only FCS of the Petri net in Figure 1(b). Note that according to the above definition the set of source transitions makes a single FCS.

3. STRUCTURAL SCHEDULABILITY ANALYSIS

In this section, we develop the theoretic foundations of our approach. Our definition of schedule is based on Petri nets. Pairwise transition dependence relation is introduced to give an intuition of the general transition dependence relation. We prove Theorem 1 and based on that give the sketch of the proof for Theorem 2.

3.1 Quasi-Static Schedules

Definition 1. *(Quasi-static schedule)*
A quasi-static schedule *of a Petri net* (N, M_0) *is a directed graph* $G(V, E)$ *satisfying the following properties:*

1. *V is finite and nonempty.*
2. *$\exists \mu : V \to R(N, M_0)$, and $\exists r \in V$ with $\mu(r) = M_0$. For any $u, v \in V, (u, v) \in E$, there exists a transition $t \in N$ such that $\mu(u)[t > \mu(v)$ [1].*
3. *If $u \xrightarrow{t_1} v$, then $u \xrightarrow{t_2} w$ if and only if there exists an FCS S such that $t_2, t_1 \in S$.*
4. *For each $v \in V$, there is a path to a vertex u, where $u \xrightarrow{t}$ for each $t \in T_S$.*
5. *For each edge (u, v) there is a cycle in G that contains (u, v).*

For the rest of paper, we use the term "schedule" to refer to a quasi-static schedule. Property 2 states that a schedule of a Petri net is a subgraph of its reachability tree. Property 3 means that if an edge leaving a node is labeled by a transition in an FCS, then this node must contain output edges for every transition in the FCS. In this case, we say that FCS is *involved* in the schedule, or the schedule *involves* the FCS. Property 3 shows that schedule considers all possible outcomes of a free choice. Property 4 denotes the fact that any input event from the environment (denoted by source transitions) is eventually fired from any vertex of a schedule. Finally, property 5 guarantees liveness of any edge of a schedule because every edge can be triggered infinitely often. Note that the last property imposes an additional requirement to a schedule definition as of compared with [7]. This confines the consideration to schedules whose graphs are strongly connected. We feel that the above requirement does not affect the applicability of our analysis in practice because it is typically a strongly connected component of a schedule which is of interest (the rest of the schedule relates to the initial part and can be preprocessed separately).

Figure 1(c) shows a schedule of the Petri net in Figure 1(b).

Starting from vertex r one can unroll a graph $G(V, E)$ representing a schedule into an acyclic tree $G'(V', E')$. The unrolling uses two mappings $origin : V' \to V$ and $instance : V' \to \mathbb{N}^2$. To relate vertexes of G' with corresponding PN markings, we assume a straightforward extension of mapping μ (from Definition 1) as $\mu(v') = \mu(origin(v'))$. Similarly we denote $v' \xrightarrow{t} u'$ iff $origin(v') \xrightarrow{t} origin(u')$ in G. The unfolding proceeds as the following:

Schedule unfolding

1. Initialize G' with $V' = E' = \emptyset$, $CurrentInstance = 0$.
2. Add root vertex r' to G' with $origin(r') = r$ and $instance(r') = 0$. Mark r' as non-leaf node.
3. For any non-leaf vertex $v' \in V'$ such that v' does not have successors in G' do:
   (a) for each edge (v, u) such that $v = origin(v')$ do
      i. Add vertex u' to V' and edge (v', u') to E'.
      ii. $CurrentInstance{+}{+}$.
      iii. Set $origin(u') = u$ and $instance(u') = CurrentInstance$.
      iv. if $\exists w' \in V'$ such that $\mu(w') = \mu(u')$ and there is a path from w' to u' in G' then mark u' as a leaf node, otherwise mark u' as non-leaf node.
4. Return the truncated unfolding.

The check for leaf node is needed to truncate the tree. As soon as in the construction of a tree one meets a vertex that is related to the same marking as some of the ancestors of this vertex in a tree, the unfolding beyond this vertex is terminated. We call the object obtained as a result of applying the above procedure a *truncated schedule unfolding.*

Lemma 1. *A truncated schedule unfolding $G'(V', E')$ for a given schedule $G(V, E)$ is finite.*

[1]Notation $u \xrightarrow{t} v$ will be used to refer at transition that fires between two schedule vertexes. We will also say that t is enabled in u.

[2]In future objects in the unfolding of a schedule graph G will be denoted by decorating the corresponding objects from a schedule with $'$.

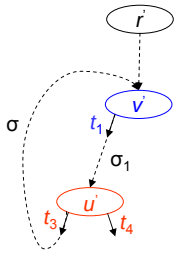


Figure 2: Illustration of the proof of Theorem 1.

Proof: The proof immediately follows from the finiteness of a schedule. One can make only a finite number of steps in the unfolding procedure before seeing the same marking repeating with some ancestor in the tree. Repeating the marking truncates the unfolding and keeps the generated prefix finite. □

3.2 Pairwise Transition Dependence Relation

DEFINITION 2. *(Pairwise transition dependence relation) A transition t of a Petri net N is said to be* dependent *on a transition t', if $\forall \boldsymbol{x} \in X$, $t \in \|\boldsymbol{x}\|$ implies $t' \in \|\boldsymbol{x}\|$. It is denoted by $t \rightarrowtail t'$.*

The pairwise transition dependence is a binary relation on T. It is reflexive and transitive, but not symmetric in general. $t \rightarrowtail t'$ means if a T-invariant contains t, it also contains t'.

THEOREM 1. *Given a Petri net N and two FCSs S_1, S_2 of N, where $S_1 = \{t_1, t_2\}$, $S_2 = \{t_3, t_4\}$, if $t_1 \rightarrowtail t_3$, $t_4 \rightarrowtail t_2$, there exists no schedule of N with S_1 or S_2 involved.*

Proof: We show that a truncated unfolding $G'(V', E')$ with root r derived by a valid schedule $G(V, E)$ with S_1 or S_2 involved, is infinite. The latter violates Lemma 1.

The proof proceeds by showing the validity of at least one of the two statements:

I1: G' contains an infinite path $r' \rightsquigarrow v'_1 \xrightarrow{t1} y'_1 \rightsquigarrow v'_2 \xrightarrow{t1} y'_2 \cdots$, such that the firing sequence corresponding to the path from r to v_k $(k = 1, 2, \dots)$ does not contain transition t_3.

I2: G' contains an infinite path $r' \rightsquigarrow u'_1 \xrightarrow{t4} z'_1 \rightsquigarrow u'_2 \xrightarrow{t4} z'_2 \cdots$, such that the firing sequence corresponding to the path from r to u_k $(k = 1, 2, \dots)$ does not contain transition t_2.

In G' let us choose vertex v' in which transitions from S_1 or S_2 are enabled and v' the closest vertex to the root r' with this property. This vertex exists because S_1 and S_2 are involved in a schedule. Without loss of generality we may assume that S_1 is enabled in v'. Then we can impose $v'_1 = v'$ in proving I1.

From Property 5 of a schedule definition and the unfolding procedure follows that $\exists w', v' \in V'$ such that the path from v' to w' contains $t1$ and $\mu(w') = \mu(v')$. This path corresponds to a firing sequence σ that makes a cycle from marking $\mu(v')$ back to itself and hence $\bar{\sigma}$ is a T-invariant. $t_1 \rightarrowtail t_3$ implies $t_3 \in \sigma$. Therefore, σ contains a node u' such that $u' \xrightarrow{t_3}$. Let u' be the closest descendant of v' with t_3 enabled.

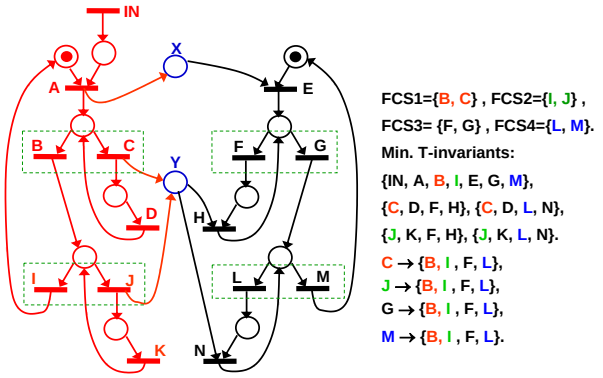


Figure 3: A Petri net contains FCSs in general cyclic dependence relation.

An illustration is shown in Figure 2.

Let us consider path $\sigma_1 \subset \sigma$ that goes from v' to u'. Two cases are possible.

Case 1. If $t_2 \in \sigma_1$ then σ_1 goes through vertex v'_2 with enabled t_2. t_1 and t_2 are from the same FCS and hence t_1 is also enabled in v'_2. Clearly the path from r' to v'_2 does not contain t_3 and therefore v'_2 satisfies the conditions of I1 and is a descendant of v'_1. Repeat the consideration for v'_2 one can conclude about the existence of infinite path satisfying I1.

Case 2. Suppose that $t_2 \notin \sigma_1$. Then the path from r' to u' does not contain transition t_2. In addition t_4 is enabled in u' (being in the same FCS as t_3) and therefore one can use u' as u'_1 in proving I2.

Bearing in mind that $t_4 \rightarrowtail t_2$ and applying the same arguments for closing the cycle from u'_1 one can conclude that there must exist a path δ from u'_1 to v'_2 in which t_2 is enabled. If δ does not contain t_3 then v'_2 satisfies the conditions of I1, which is the basis for constructing an infinite path. If δ contains t_3 then by choosing the first firing of t_3 in δ, one can obtain a vertex u'_2 in which t_3 is enabled together with t_4, and u'_2 is a descendants of u'_1. This proves I2. □

3.3 General Transition Dependence Relation

In this session, the dependence relation is generalized, and a weaker sufficient condition to prove unschedulability is proposed.

DEFINITION 3. *(General transition dependence relation) A transition t of a Petri net N is said to be* dependent *on a set S of transitions, if $\forall \boldsymbol{x} \in X$, $t \in \|\boldsymbol{x}\|$ implies $\exists t' \in S$ such that $t' \in \|\boldsymbol{x}\|$. It is denoted by $t \rightarrowtail S$.*

Our previous definition of pairwise transition dependence relation can be viewed as a special case of the general dependence relation with $|S| = 1$. Note that by definition the relation is monotonically non-decreasing, meaning if $t \rightarrowtail S$, then $\forall S', S \subseteq S'$, $t \rightarrowtail S'$. Two trivial cases of general dependence relation are $t \rightarrowtail \{t\}$ and $t \rightarrowtail T$. Figure 3 shows a Petri net that contains transitions in a general dependence relation.

DEFINITION 4. *(Cover of FCSs) Given a set of FCS $\mathbb{S} = \{S_1, S_2, \dots, S_n\}$, a set of transitions $\mathbb{S}^c = \{t_1, t_2, \dots, t_n\}$ is said to be a* cover *of $\mathbb{S}$, if $t_i \in S_i, i =$*

$1, 2, \ldots, n$. *For a given* $\mathbb{S}^c$ *denote* $\overline{\mathbb{S}^c} = \{t \mid \exists S \in \mathbb{S}, t \in S$ *and* $t \notin \mathbb{S}^c\}$.

A cover of a set of FCSs contains exactly one transition from each FCS. Note that the number of covers of a set of FCSs is exponential in the number of FCSs. In Figure 3, $\mathbb{S}^c = \{C, J, G, M\}$ is a cover of $\{FCS1, FCS2, FCS3, FCS4\}$, and $\overline{\mathbb{S}^c} = \{B, I, F, L\}$.

DEFINITION 5. *(Cyclic dependence relation) There exists a* cyclic dependence relation *among* $\mathbb{S}$*, a set of FCSs of a Petri net* N*, if* $\exists \mathbb{S}^c$*, such that* $\forall t \in \mathbb{S}^c, t \rightarrowtail \overline{\mathbb{S}^c}$.

Note that although the general dependence relation is monotonic, the cyclic dependence relation is not monotonic in general. It means that we can not prove the existence of the relation among a set of FCSs by proving the existence of the relation among its subset and vice versa.

THEOREM 2. *Given a Petri net* N *and a set* $\mathbb{S}$ *of FCSs of* N*, if there exists a cyclic dependence relation among* $\mathbb{S}$*, then there exists no schedule of* N *with any FCS in* $\mathbb{S}$ *involved.*

Sketch of proof: The proof can be done similar to the proof of Theorem 1. One can show that a truncated unfolding $G'(V'E')$ obtained by a schedule $G(V, E)$ is infinite. This could be done by proving that G' contains an infinite path started from root r', such that the path does not contain any $t \in \mathbb{S}^c$. □

There are several observations about the theorem. First, it provides a sufficient condition to prove unschedulability. Second, the condition does not involve the initial marking of a Petri net. It is a structural property of a Petri net. Third, the theorem can be applied to general Petri nets.

4. CHECKING CYCLIC DEPENDENCE RELATION

Although our theory is built upon the set of all T-invariants of a Petri net, interestingly enough, our algorithm does not need to compute such a set or any generating set of all T-invariants. In this section, we propose a linear programming based algorithm to check the existence of a cyclic dependence relation among a given set of FCSs.

Given a Petri net N, its incidence matrix A, and a set $\mathbb{S}$ of FCSs of N to be checked, the algorithm iterates through all possible covers of $\mathbb{S}$ till one cover leads to a dependence relation. For each cover, a feasibility problem of linear programming is constructed. As proved in Theorem 3, a solution to the feasibility problem provides a counterexample to the dependence relation. If no solution is found, the dependence relation holds. Note that whether a cover $\mathbb{S}^c$ leads to a cyclic dependence relation among $\mathbb{S}$ can be checked in polynomial-time.

THEOREM 3. *Given a Petri net* N *and its incidence matrix* $\boldsymbol{A}$*, a transition* t_i *is dependent on a set* S *of transitions if and only if* $\nexists \boldsymbol{x} \in \mathbb{R}^{|T|}$ *such that* $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{x} = 0, \boldsymbol{x} \geq 0, x_i > 0, \forall t_j \in S, x_j = 0$.

Proof "⇒": If $t_i \rightarrowtail S$ does not hold, by definition, there exists a T-invariant $\boldsymbol{x} \in \mathbb{N}^{|T|}$, such that $x_i \geq 1, \forall t_j \in S, x_j = 0$.

"⇐": Since the incidence matrix $\boldsymbol{A}$ is an integer matrix, if there exists a real vector that satisfies all the constraints, then there exist a rational vector $\boldsymbol{x}$ also satisfying the constraints. Let θ be a common multiple of all the denominators of the elements of $\boldsymbol{x}$ and let $\boldsymbol{x}' = \theta\boldsymbol{x}$. By definition, $\boldsymbol{x}'$ is a T-invariant, and $\boldsymbol{x}' \geq 0, x'_i > 0, \forall t_j \in S, x'_j = 0$. □

Algorithm 1 Checking cyclic dependence relation using linear programming

INPUT: A: the incidence matrix of a Petri net, $\mathbb{S}$: the set of FCSs to be checked.
OUTPUT: returns TRUE if there exists a cyclic dependence relation among $\mathbb{S}$, FALSE otherwise

```
for all cover S^c of S do
  dependent ⇐ TRUE
  for all t_i ∈ S^c do
    LP ⇐ A^T x = 0 ∩ x ≥ 0
    LP ⇐ LP ∩ x_i > 0
    for all t_j ∈ S̄^c do
      LP ⇐ LP ∩ x_j = 0
    end for
    if LP is feasible then
      dependent ⇐ FALSE
      break
    end if
  end for
  if dependent = TRUE then
    return TRUE
  end if
end for
return FALSE
```

5. EXPERIMENTS

In this Section we show that the sufficient condition introduced in Theorem 2 holds for a wide class of real-life industrial applications. It means that Petri nets generated from their system specifications are not schedulable, and our approach can be effectively applied to establish that. Note that to prove unschedulability of a Petri net based on Theorem 2, we also need to assert that each schedule involves at least one FCS of the Petri net. We use some public available JPEG and MPEG codecs as our test bench. The codecs used in our experiments are modelled as Kahn process networks.

5.1 Benchmarks

MPEG-2 decoder. We use an MPEG-2 decoder [17] developed by Philips Research Laboratories. The decoder is written in 5,000 lines of YAPI [9] code, a system specification language based on C++. As shown in Figure 4, the system consists of 11 concurrent processes communicating through 45 channels. We perform schedulability analysis on 5 processes: *TdecMV*, *Tpredict*, *Tisiq*, *Tidct*, and *Tadd*. The first two processes implements the spatial compression decoding. The last three processes implements the temporal compression decoding and image generation. In total, the 5 processes have 10 channels, and 13 interfaces (communicating ports with other processes or the environment).

M-JPEG encoder.* We use an M-JPEG* [15] encoder also developed by Philips. The source code is obtained through the Sesame [1] project public release. The encoder is written in about 2,000 lines of YAPI code. As shown in Figure 5,

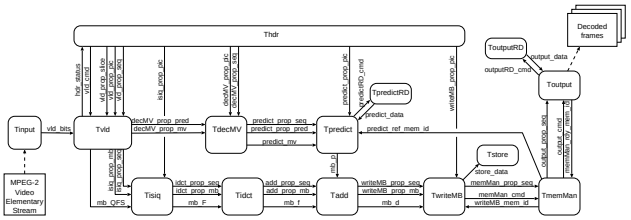


Figure 4: An MPEG-2 decoder modeled as a KPN.

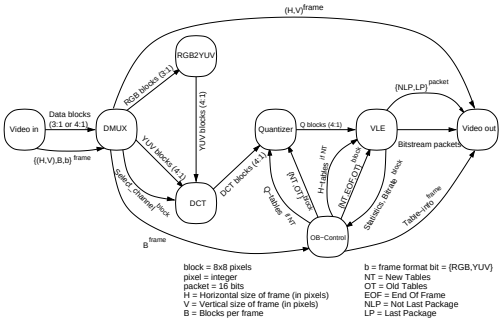


Figure 5: An M-JPEG* encoder modeled as a KPN.

the system consists of 8 processes communicating through 18 channels. Video data are parsed by Process *DMUX* and are sent directly to Process *DCT* or via Process *RGB2YUV*, depending on the video format. Video parameters are send to Process *OB-Control*, which controls the video processing in Process *DCT*, *VLE*, and Vudeo Out. It also collects statistics information to adjust Huffman coding tables and quantization tables. We perform schedulability analysis on the entire system.

XviD MPEG4 encoder. Our model of a XviD MPEG4 encoder is based on the KPN described in [5] and the C source code from [3]. It was originally developed as a Sesame application. The encoder supports two frame types: I-frame and P-frame. It performs motion estimation analysis to determine if an incoming frame will be treated as I-frame or P-frame. Consequently two types of frame will go through different processing paths. An I-frame will be split into macro-blocks and encoded independently. In a P-frame, a macro-block could be an intra-block, an inter-block, or an not-coded-block, depending the value of Sum of Absolute Differences (SAD). The granularity of tokens passing between processes is macroblock. The system consists of 15 processes with 40 channels.

PVRG JPEG encoder. We obtain the Stanford Portable Video Research Group (PVRG) JPEG codec source code from [2]. Based on JPEG baseline standard, our model consists of 10 processes and 21 channels. The functional core part consists of 4 processes implementing Discrete Cosine Transform (DCT), quantization, Huffman coding, and control. The granularity of tokens passing between processes is block. We preform schedulability analysis both on Petri nets generated from the model of the encoder and its functional core.

Instance	Place	Tran.	Arc	FCS	Runtime	
					Analyzer	Scheduler
pJPEGe1	22	22	52	4	0.04s	>24hr
pJPEGe2	26	28	64	6	0.27s	>24hr
pJPEGe3	26	27	64	6	0.44s	>24hr
pJPEGe4	67	68	167	14	0.54s	>24hr
pMJPEGe	100	98	278	17	0.75s	>24hr
pMPEG2d	68	76	176	18	1.67s	>24hr
xMPEG4e	72	72	184	15	0.38s	>24hr

Table 1: Statistics of schedulability analysis of Petri net models of JPEG and MPEG codecs

We also experiment different modelling decisions in this test case. For example, the control of synchronization between concurrent processes could be managed by a single master process, or distributed among processes.

5.2 Results

We implement our quasi-static schedulability analyzer in C. All experiments are run on a 3.06 GHz Intel Xeon CPU with 512 KB cache and 3.5 GB memory. Since to the best of our knowledge there is no other quasi-static schedulability analyzer available, we compare performance with a quasi-static scheduler. The scheduler performs a schedulability analysis via heuristic construction of a schedule. Based on the test cases, our analyzer typically proves a Petri net unschedulable within one second, while the scheduler fails to terminate in 24 hours. Details of the experiments are summarized in Table 1.

Our schedulability analyzer is effective because there exist certain program structures in the codecs. We illustrate it using a Huffman coding process of a JPEG encoder. Figure 6 shows a simplified description of the process. The process first reads from a control process a header which includes all parameters necessary to perform Huffman coding on a block. Then it iterates through all blocks of a component in a Minimum Coded Unit (MCU). The Huffman coding and reading from a zigzag process are performed at the block level inside the loop. Since JPEG standard requires samples of a component must use the same Huffman coding table, and multiple component samples could be interleaved within a compressed data stream, it needs to update the Huffman table from time to time. Also note that the loop has a variable number of iterations. The vertical and horizontal sampling factors could be different for different components and only known at run-time. All the above requires communications inside and outside a loop structure. The quantization process has a similar program structure to the Huffman coding process, because samples of a component are required to use the same Quantization table and processing and communication data is performed at block level. The control process synchronizes the two concurrent processes such that a block is processed in the quantization process and later in a Huffman coding process with the set of parameters (e.g. vertical and horizontal sampling factors) of the same component.

Synchronized communications inside and outside a loop structure are common in the codecs. However, the controls of loops are abstracted as non-deterministic free choices in a Petri net. These two factors cause unschedulability which can be checked efficiently by our approach.

Note that our schedulability analyzer also computes the minimum set of FCSs that has a cyclic dependence relation,

```
PROCESS Huffman(
              In_DPORT Control_headerIn,
              In_DPORT Zigzag_blockIn,
              Out_DPORT Output)
{
  while(1) {
    READ_DATA(Control_headerIn, header, 1);
    Vi = getVSF(header);
    Hi = getHSF(header);
    Htable = getHtable(header);
    for(v=0; v<Vi; v++) {
      for(h=0; h<Hi; h++) {
        READ_DATA(Zigzag_blockIn, block, 1);
        block = HuffmanEncoding(block, Htable);
        WRITE_DATA(Output, block, 1);
} } } }
```

Figure 6: A simplified Huffman coding process described in FlowC.

once it proves a Petri net is unschedulable. The minimum set of FCSs provides useful information to find the correlated control structures.

6. LIMITATIONS

We rely on our experience with JPEG MPEG codecs and comparison with a scheduler to claim the effectiveness and efficiency of our approach. However, various issues limit the applicability of our approach. In this section, we discuss some of them and establish directions for the future work.

Dependence relation driven by firability. We defined a dependence relation between transitions statically based on the existence of T-invariants. However, not every T-invariant may be firable in a Petri net. If the dependency relation is violated because of non-firable T-invariants, our approach would fail to establish unschedulability. Taking into account firability of T-invariants would increase the resolution power of the proposed method.

Efficiency of the analysis. Although each dependence check can be done in polynomial time, the total number of checks to prove unschedulability is exponential in the number of FCSs. Currently we are studying the conditions based on checking the rank of the incidence matrix [10], which could lead to more efficient means to prove or disprove schedulability.

Necessity of the conditions for unschedulability. Theorem 1 and 2 provide only sufficient conditions for unschedulability. One possible research direction is to look for subclasses of Petri nets for which these conditions are also necessary.

7. CONCLUSION

We introduced a Petri net structural property, and prove that if the property holds among the set of FCSs of a Petri net, there exists no quasi-static schedule. We also present an algorithm based on linear programming to check if such a property holds. We performed schedulability analysis on Petri nets generated for several benchmark examples from multi-media applications. The experiment results show that our approach is valid and effective.

8. REFERENCES

[1] Sesame project public release. URL: `http://sesamesim.sourceforge.net`.

[2] Stanford PVRG JPEG codec. URL: `http://www.dclunie.com/jpegge.html`.

[3] XviD MPEG-4 video codec. URL: `http://www.xvid.org`.

[4] G. Arrigoni, L. Duchini, L. Lavagno, C. Passerone, and Y. Watanabe. False path elimination in quasi-static scheduling. In *Proceedings of the Design Automation and Test in Europe Conference*, March 2002.

[5] P. Broekhof, N. Roosen, J. Verhoef, and W. Jun. Modeling XviD as a Kahn process network, a Sesame application design document. URL: `http://staff.science.uva.nl/~andy/apps/xvid.pdf`.

[6] J. T. Buck. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*. PhD thesis, University of California, Berkeley, 1993.

[7] J. Cortadella, A. Kondratyev, L. Lavagno, C. Passerone, and Y. Watanabe. Quasi-static scheduling of independent tasks for reactive systems. In *Proceedings of the 23rd International Conference on Applications and Theory of Petri Nets*, pages 80–100, 2002.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symp. on Principles of Programming Languages*, Los Angeles, January 1977.

[9] E. de Kock, G. Essink, W. Smits, P. van der Wolf, J.-Y. Brunel, W. Kruijtzer, P. Lieverse, and K. Vissers. YAPI: Application Modeling for Signal Processing Systems. In *Proceedings of the* 37^{th} *Design Automation Conference*, June 2000.

[10] J. Desel. Private communication, August 2004.

[11] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[12] G. Kahn. The semantics of a simple language for parallel programming. In *Information processing*, pages 471–475, Aug 1974.

[13] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *Information processing*, pages 993–998, Aug 1977.

[14] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow graphs for digital signal processing. *IEEE Transactions on Computers*, Jan. 1987.

[15] P. Lieverse, T. Stefanov, P. van der Wolf, and E. Deprettere. System level design with spade: an m-jpeg case study. In *Proceedings of IEEE/ACM International Conference on Computer Aided Design*, pages 31–88, Nov 2001.

[16] T. Murata. Petri nets: properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr. 1989.

[17] P. van der Wolf, P. Lieverse, M. Goel, D. Hei, and K. Vissers. An MPEG-2 Decoder Case Study as a Driver for a System Level Design Methodology. In *Proceedings of the* 7^{th} *International Workshop on Hardware/Software Codesign*, May 1999.