1989

Critical Path Analysis of Real-Time ADA Programs

Dan C. Marinescu

Ryan Stansifer

Report Number: 89-855
CRITICAL PATH ANALYSIS OF
REAL-TIME ADA PROGRAMS

Dan C. Marinescu
Ryan Stansifer

CSD-TR-855
February 1989
Revised May 1989
APPLICATION OF PETRI NETS TO CRITICAL PATH ANALYSIS OF REAL-TIME ADA PROGRAMS

Dan C. Marinescu
Ryan Sransifer

Computer Sciences Department
Purdue University
West Lafayette, IN 47907, USA
1. INTRODUCTION

Real-time systems play a vital role in our society. Applications for real-time systems are encountered in areas like industrial control and automated manufacturing systems, data acquisition and analysis systems in high energy physics, air traffic control, defense, etc.

In a real-time system there are end-to-end timing constraints for various activities in the system. Due to timing requirements the design and implementation of real-time systems reveal new and very difficult problems which were not previously encountered for other classes of systems [5,11].

In this paper, we propose a methodology for testing real-time programs written in Ada. Our approach is based on translation of Ada programs into Petri nets and on determining a critical path in the program execution through the analysis of the Petri net model of the program.

Correctness of real-time programs has two dimensions, functionality and timing. A real-time program which performs its required functions, but fails to meet required deadlines is considered to be incorrect. Consequently, testing of real-time programs has two aspects: testing of functionality and verification of timing requirements. For the most part, testing of the functionality of real-time programs is performed using methodologies known from the testing of non-real time programs. Then the execution time of the program is measured for various test cases to determine if the timing constraints are met.

This is unsatisfactory for several reasons:

1. Measurements can be made only after the program is developed. It is not possible to identify potential problems in the early stages of the design process.

2. Precise timing measurements are generally difficult to be carried out. Software probes alter the actual timing of events in the system to be measured. The execution time of any code is data dependent and it is affected by hardware and software errors and retries. A measurement has to be repeated several times and statistical methods to determine confidence intervals have to be used.

3. It does not accommodate changes in the specifications. In a real-time system there are end-to-end timing constraints and every time a change in any sub-system is made, there are ripple effects propagating through the entire system. For this reason, often it is easier to re-design the entire system rather than accommodate a minor change in the specifications.

4. Any change in the characteristics of the hardware or of the system software invalidates the timing measurements performed on real-time applications.

5. The verification effort grows very fast when the problem size and program complexity increases. Most of the real-time system designed today are very sophisticated, often they perform some forms of knowledge processing, e.g., target identification.

Our goal is to design a methodology for verification of timing correctness of real-time programs applicable to large real-time software development projects.

Clearly, the execution time of a real-time program depends upon the hardware and the operating system environments. To avoid the hard problems related to scheduling, we consider at first a simplified model of the environment supporting the execution of the Ada program. We assume that whenever a new task is created a process to execute the task is available immediately. Moreover, whenever a task begins execution, it is not interrupted by events other than those explicitly specified in the program itself. For example, the model does not take page faults into account. While these assumptions may seem restrictive, they are necessary to get meaningful timing estimates that are independent of the environment.
The immediate target of our effort is directed toward verification of timing constraints of programs written in Ada, but the methodology can be easily extended to other programming environments for real-time systems for example MODULA, microcode, etc. There are numerous real-time applications written in Ada and this motivates our emphasis of this particular programming environment. Our goal is to develop an approach for verification with the following characteristics:

1. It should be capable of providing early indication of timing correctness. In the early stages of the program design, it operates using timing estimates and as actual programs become available more accurate data from the actual program replaces the estimates.

2. The verification methodology should match the abstraction and refinement process involved in the design of any software system. It should be simple to accommodate changes in program specification.

3. The verification process should be largely automated.

4. The verification methodology should be largely machine independent. The hardware and systems software characteristics are kept in a machine-dependent database used to translate machine independent cost functions into timing estimates for a particular hardware and software environment.

The programming language Ada [13] is a logical choice for programming real-time systems. It is one of the few high-level languages with constructs for specifying the concurrent execution of program parts and for specifying time dependent actions. An Ada program may contain several processes called tasks. Each task specifies a particular sequence of actions to be performed one after another, but several tasks may execute concurrently. Tasks may synchronize and communicate with one another through that is called rendezvous. As in CSP it is possible to request communication with a number of tasks and select some task that is ready to communicate. It is possible to model time-out by withdrawing the willingness to rendezvous after a certain interval of time.

In Europe, where the study of the formal semantics of Ada is quite advanced, some attempts to use Petri nets to capture the real-time semantics of Ada have been made [4]. We will make use of these efforts in devising the best Petri net models of Ada programs. At the University of Illinois, Chicago, a tool has been developed to translate Ada to Petri nets [7,10], although this does not take into account real-time constructs.

This paper is organized as follows: Section 2 presents our approach in translating Ada programs into Petri net models. Section 3 introduces a formal approach to critical path analysis and Section 4 illustrates these concepts through an example.

2. PETRI NET MODELS OF ADA PROGRAMS

In this section we briefly sketch how we obtain Petri net models of Ada programs.

The following Ada program is a simple example of a program using tasks. It does not involve any synchronization. Several tasks are started up during the program execution. These tasks represent separate threads of control which may be executed simultaneously with the “main” thread of control.

```
procedure P is
  task T1;
  task type TT2;
  type PrTT2 is access TT2;  -- pointer to task type
```
X : PtrTT2;  -- declare a pointer to a task

begin -- at this point all tasks declared at this level begin
S_3;
X := new TT3;  -- A task of type TT3 begins here
S_4;
B := declare
  task T3;
  task body T3 is begin S_5 end T3;
begin
  S_6;
end B;
S_7;
end P;

The definition of the Ada language specifies some constraints on the order in which the statements in the program can be executed. In the absence of rendezvous these requirements form a partial order. The partial order for the package P is shown in the Hasse diagram of Figure 1.

Any total ordering of the partial order represented in Figure 1 is a possible sequence of statements that a uniprocessor might choose to execute the program P. By enumerating all total orders, one arrives at all the different choices the run-time system might make in the execution of the program. In testing the program one would want to examine all possible orders. The difficulty is that some particular computing environment might not ever choose some of the paths. In practice, it may be impractical to test all the paths. In which case a "randomly" generated collection of paths is desirable.

Another representation of the semantics of program P is the Petri net in Figure 4. Any firing sequence of this Petri net is a possible execution of the statements in program P. Petri nets are indispensable in representing the semantics of Ada program with rendezvous.

We look at another example to indicate the difficulties raised by rendezvous. The following Ada programs has a task with one entry point, but this entry is accepted at two separate parts of the task.
task Task 1 is
  entry Rendez;
end Task 1;
task body Task1 is
begin
  \textit{S}_1;
accept Rendez do -- accept statement 1
  S2;
end Rendez;
S3;
accept Rendez do -- accept statement 2
  S4;
end Rendez;
S5;
end Task 1;

task Task2;
task body Task2 is
begin
  S6;
  Task1.Rendez; -- call entry 1
  S7;
  Task1.Rendez; -- call entry 2
  S8;
end Task2;

We know by examination of the program that entry call 1 will rendezvous with accept
statement 1, and entry call 2 will rendezvous with accept statement 2. But in general, it is not
known which call will be accepted by which accept statement.

The solution to this problem is to create a specialized Petri net to handle the synchroniza­
tion. For each call site, we must have a synchronization subnct for every possible accept state­
ment. In the example above, this results in \(2 \times 2 = 4\) subnets.

It is most convenient to give the formal translation to Petri nets using the incidence matrix
rather than some graphical representation. We have developed an algorithm to translate a subset
of Ada into the incidence matrix of a connected, pure Petri net. This algorithm follows the
form of a semantic function in the literature of denotation­
semantics [12] and has been imple­
mented in the language ML [6].

Here is a small and incomplete fragment of the translation algorithm for a basic atomic action,
an accept statement and an entry call. Notice that the function StmtSem is a higher-order func­
tion that translates the abstract syntax of Ada into a list of list representing the incidence matrix.

fun
StmtSem env (Basic a) b e = [(st a, [(b,-1), (e, 1)])]
StmtSem env (Accept(y,n,sl)) b e =
(*
  This subnet is disconnected; b and e are ignored. When entry call
  made, function Subnet must find P_IN and P_OUT and hook us in.
*)
let
  val P_IN = rl(y,n);
  val P_OUT = r2(y,n);
in
  Comp (map (StmtSem env) sl) P_IN P_OUT
end
The function Comp composes statements and the function Subnet creates the subnet for rendezvous. The program works by creating a fragment of the Petri net when given the single entry and single exit point, the formal parameters b and e in the ML program, to the subnet. This works for all statements except the accept and entry statements. These statements require additional arcs which are patched up using the environment.

3. A FORMALISM FOR CRITICAL PATH ANALYSIS OF REAL-TIME ADA PROGRAMS

In this section we introduce a formalism for critical path analysis of real-time Ada programs. First we define Ada systems which are Petri net models of real-time Ada programs, then we present an algorithm for computing the critical path. Our net models differ from the ones used by Murata [7] because we need to account for the time required by different activities in a real-time system. The ultimate goal is to reduce a given Ada system to a canonical form which consists of two places and one transition with a cost equal to the cost of the critical-path.

**Definition**: Cycle, Path, Acyclic net.

Given the net \( N = (S, T; F) \) a cycle is a sequence \( (x_0, x_1, \ldots, x_m) \) with the following properties:

\( (a) \ x_i \in S \cup T, \ 0 \leq i \leq m, \)
\( (b) \ (x_i, x_{i+1}) \in F, \ 0 \leq i < m, \) and
\( (c) \ x_m = x_0. \)

A path is sequence \( (x_0, x_1, \ldots, x_m) \) with properties (a), (b) and (d)

\( (d) \ \forall i, j \ 0 \leq i \leq m, \ 0 \leq j \leq m, \ i \neq j \Rightarrow x_i \neq x_j. \)

A net \( N \) is acyclic iff no cycles exist in the net.

**Definition**: Finite, Pure, Connected Petri Net.

A Petri Net \( N = (S, T; F) \) is finite iff

\[ |S| < \infty, \ |T| < \infty. \]

The net is pure iff

\[ \forall t_k \in T, \ |^* t_k | \geq 1. \]

The net is connected iff

\[ \forall t_k \in T, \ ^* t_k \cap \ ^* t_k = \emptyset, \text{ and } \forall P_i \in P, P_i \neq P_{IN}, \ |^* P_i | \geq 1. \]

**Definition**: Ada system.

An Ada system \( A = (S, T; F, M_{IN}) \) is a finite, pure, connected Petri net with the initial marking \( M_{IN} \).

In addition, every transition of an Ada system has a non-negative cost associated with it

\[ \forall t_i \in T, \ C(t_i) = \tau_i \text{ with } \tau_i \geq 0. \]
The underlying net of \( A \), \( A = (S, T; F) \) has a source place \( P_{IN} \), a source transition \( t_{IN} \), a sink place \( P_{OUT} \) and a sink transition \( t_{OUT} \).

An Ada system with no cycles will be called an *acyclic Ada system*. In this paper we are concerned with acyclic Ada systems.

**Definition:** Initial and final markings of an Ada system.

The *initial marking* of an Ada net is

\[
M_{IN} = (1, 0, \ldots, 0, \ldots, 0, 0)
\]

with only one token in \( P_{IN} \). The *final marking* of an Ada net is

\[
M_{OUT} = (0, 0, \ldots, 0, \ldots, 0, 1)
\]

with only one token in \( P_{OUT} \). Note that \( M_{OUT} \) is a dead marking.

**Definition:**

An Ada system \( A = (S, T; F, M_{IN}) \) terminates if

\[
M_{OUT}
\]

is the only dead marking of \( A \).

**Definition:** Execution path.

An *execution path* of an Ada system is a path

\[
e_i = (x_0^{(0)}, x_1^{(0)}, \ldots, x_m^{(0)})
\]

with \( x_0 = P_{IN} \).

Let \( t_{IN}, t_1^{(0)}, \ldots, t_q^{(0)} \) be all the transitions in the execution path \( e_i \). Sometimes it is convenient to refer to an execution path by enumerating only the transitions in \( e_i \)

\[
e_i = (t_{IN}, t_1^{(0)}, t_2^{(0)}, \ldots, t_q^{(0)})
\]

**Observation:**

In an acyclic Ada system all elements of an execution path are distinct

\[
\forall x_\alpha \in e_i \text{ and } \forall x_\beta \in e_i, \quad x_\alpha \neq x_\beta \text{ if } \alpha \neq \beta.
\]

**Definition:** Execution sequence of an Ada system.

\[
ES = (M_{IN}, M_1, \ldots, M_q, t_{IN}, t_1, \ldots, t_{q-1})
\]

\( ES \) is a double sequence such that for all \( t_j, M \rightarrow t_j \rightarrow M' \) is the operation of firing transition \( t_j \) and changing marking \( M \) into its follower marking \( M' \).

It is known that the sequence of transitions and the initial marking uniquely determine the sequence of markings. Call \( ES(M_{IN}) \) the set of all execution sequences \( \sigma_i \) of an Ada system.

An execution sequence \( \sigma_k \) is said to be a *proper execution sequence* iff \( M_q = M_{OUT} \) and \( t_{q-1} = t_{OUT} \).
Definition: Feasible execution path.

An execution path \( e_i = (x_0^{(i)}, x_1^{(i)}, \ldots, x_n^{(i)}) \) of an Ada system is a feasible execution path iff every element \( x_j^{(i)} \in e_i, \ 0 \leq j \leq n \) belongs to some execution sequence \( \sigma_k \).

Definition: The cost associated with an execution path \( e_i \).

Let \( e_i \) be an execution path of an Ada system, \( A \),
\[
e_i = (t_{IN}, t_1^{(i)}, \ldots, t_q^{(i)})
\]
Then
\[
c_i = \mathcal{C}(e_i) = \sum_{\text{all } t_j^{(i)} \in e_i} \mathcal{C}(t_j^{(i)})
\]

Definition: Critical execution path of an Ada system.

Let \( E = \{e_1, \ldots, e_r\} \) be the set of all feasible execution paths that include \( t_{OUT} \) and \( P_{OUT} \) in an Ada system \( A \). Let \( c_i \) be the cost of \( e_i \), \( c_i = \mathcal{C}(e_i) \).

Let
\[
c_{\text{critical}} = \max (c_1, c_2, \ldots, c_r)
\]
Let \( e_j \) be the execution path such that
\[
c_{\text{critical}} = \mathcal{C}(e_j)
\]
Then we call \( e_j \) the critical execution path and denote it by \( e_{\text{critical}} \).

Definition: Cost-equivalent Ada systems.

Two Ada systems are cost-equivalent if their critical paths have equal costs.

Definition: Cost-equivalent canonical form of an Ada system.

Given an Ada system \( A \) having a critical path with cost \( c_{\text{critical}} \) the cost-equivalent canonical form of the Ada system \( A \) is an Ada system \( A_R \) with two places, \( P_{IN} \), and \( P_{OUT} \), and one transition, \( t_{\alpha} \) with cost \( \mathcal{C}(t_{\alpha}) = c_{\text{critical}} \) as shown in Figure 2.

\[\begin{align*}
&\text{Figure 2. The cost-equivalent canonical form of an Ada system.}
\end{align*}\]
Definition: Cost of reaching a place.

Let $A$ be an acyclic Ada system and let $P_j \in P$, $P_j \neq P_{IN}$. The cost of reaching the place $P_j$ is defined as

$$\Gamma(P_j) = \max_{t_k \in T_{P_j}} \left( \tau_k + \max_{P_m \in T_{P_j}} \Gamma(P_m) \right)$$

with $\tau_k = \sigma(t_k)$ the cost associated with transition $t_k$.

Definition: The cost of reaching a place via a transition.

Let $A$ be an acyclic Ada system and let $P_j \in P$ and $t_k \in T$, $t_k \in T_{P_j}$. The cost of reaching $P_j$ via $t_k$ is defined as

$$\Gamma(P_j/t_k) = \tau_k + \max_{P_m \in T_{P_j}} \Gamma(P_m)$$

Note that

$$\Gamma(P_j) = \max_{t_k \in T_{P_j}} \Gamma(P_j/t_k)$$

If $|P_j| = 1$ and $t_k \in T_{P_j}$ then

$$\Gamma(P_j) = \Gamma(P_j/t_k).$$

Definition: The cost matrix of an acyclic Ada system.

Let $A = (S,T,F,M_{IN})$ be an acyclic Ada system with incidence matrix $N = \left[ N(P_i,t_j) \right]$, and $n = |S|$, $m = |T|$.

The cost matrix $\Delta$ of $A$ is an $n \times m$ matrix with elements defined by

$$\Delta(P_i,t_j) = \Gamma(P_j/t_k) \quad \text{iff} \quad N(P_i,t_j) > 0$$

$$\Delta(P_i,t_j) = \Gamma(P_i) \quad \text{iff} \quad N(P_i,t_j) < 0$$

$$\Delta(P_i,t_j) = \infty \quad \text{iff} \quad N(P_i,t_j) = 0$$

Algorithm 1: Computes the cost of reaching places in an acyclic Ada system that terminates.

Step 1:

Set the cost of reaching $P_{IN}$, $\Gamma(P_{IN}) = 0$. Divide the set of places into two disjoint subsets $P_1$ and $P_2$, $P = P_1 \cup P_2$ and $P_1 \cap P_2 = \emptyset$ by choosing $P_1 = \{P_{IN}\}$ and $P_2 = P - \{P_{IN}\}$. Divide the set of transitions into two disjoint subsets $T_1$ and $T_2$, $T = T_1 \cup T_2$ and $T_1 \cap T_2 = \emptyset$ by choosing $T_1 = \emptyset$ and $T_2 = T$.

Step 2:

Find a transition $t_k \in T_2$ such that all places in its preset are in $P_1$.

$$\forall P_j \in T_k \text{ then } P_j \in P_1.$$
\[ \Gamma(t_k) = \max_{\forall P_j \in \mathcal{J}_1} \{ \Gamma(P_j) \} \]

Set
\[ \mathcal{J}_1 = \mathcal{J}_1 + \{ t_k \} \]

and
\[ \mathcal{J}_2 = \mathcal{J}_2 - \{ t_k \} \]

For every \( P_t \in t_k \) compute \( \Gamma(P_t/t_k) = t_k + \Gamma(t_k) \). Then for every \( P_t \in t_k \) determine if all \( t_q \in P_t \) are in \( \mathcal{J}_1 \). If so, compute \( \Gamma(P_t) = \max_{\forall t_q \in P_t} \{ \Gamma(t_q) \} \) and set
\[ \mathcal{O}_1 = \mathcal{O}_1 + \{ P_t \} \]
\[ \mathcal{O}_2 = \mathcal{O}_2 - \{ P_t \} \]

If \( \mathcal{J}_2 \neq 0 \) repeat Step 2, otherwise terminate.

Observation:
This algorithm assigns the cost of reaching a place \( P_j \) via transition \( t_k \) at the time when \( t_k \) is enabled and fires. If the Ada system \( \mathcal{A} \) terminates, then at that time, all places in the preset of \( t_k \) have been assigned costs of reaching. At the time transition \( t_{OUT} \) fires \( \mathcal{J}_2 = \emptyset \) and \( P_{OUT} \) is assigned a cost of reaching hence \( \mathcal{O}_2 = \emptyset \) and the algorithm terminates.

Theorem 1:
Algorithm 1 assigns costs of reaching to place \( P_i \in P \) of an Ada system iff \( P_i \) belongs to a feasible execution path.

Proof:
Let \( P_i \) be a place that belongs to the feasible execution path \( e_q \). This means that there is an execution sequence \( \sigma_P \) which contains markings \( M' \) and \( M'' \) as well as transitions \( t_k \) such that \( M' \rightarrow t_k \rightarrow M'' \) and \( \Gamma(t_k) = 0 \) and \( M''(P_i) > 0 \).

According to the previous observation at the time transition \( t_k \) fires, we are able to compute the cost of reaching \( P_i \) via \( t_k \).

If \( P_i \) does not belong to any execution sequence, then no transition in its preset is ever enabled and we are not able to compute the cost of reaching \( P_i \).

If multiple feasible execution path contain \( P_i \) then the cost of reaching \( P_i \) can be computed at the time the last transition \( t_k \in P_i \) is enabled.

Theorem 2:
The cost of reaching the place \( P_{OUT} \) in an Ada system with feasible execution paths to \( P_{OUT} \) is the cost associated with the critical execution path.

Proof:
If the system has feasible execution paths then let \( e_j \) be the critical execution path which contains \( P_{OUT} \) as well as \( t_{OUT} \). Let \( P_q \in t_{OUT} \) be the place with the largest cost of reaching in the preset of \( t_{OUT} \). Clearly \( P_q \) has been used by the algorithm in computing the cost of reaching \( P_{OUT} \).

But \( P_q \) belongs to the critical execution path. Suppose that this is not true and there is another place \( P_{q'} \in t_{OUT} \) which belongs to the critical execution path. Since \( \Gamma(P_{q'}) < \Gamma(P_q) \),
the execution path containing \( P_{q'} \) is not the critical path and belongs to the critical path. The reasoning is then repeated for all places until \( P_{IN} \) is reached.

Theorem 2 shows that Algorithm 1 can be used to reduce an Ada system to its cost-equivalent canonical form.

The critical path of an Ada system can be computed using the cost matrix. The following algorithm allows the construction of \( \Delta \).

Algorithm 2: Construct the Cost Matrix

Step 1:
Set \( \Gamma(P_{IN}) = 0 \) and \( \Gamma(t_{IN}) = 0 \)

\[ \mathcal{P}_1 = \{ P_{IN} \} \quad \mathcal{T}_1 = \{ T_{IN} \} \]

Step 2:
Find all transitions \( t_k \) such that \( \bullet t_k \subseteq \mathcal{P}_1 \). For all \( P_j \in \bullet t_k \) set

\[ \Delta(P_j, t_k) = \Gamma(P_j) \]

Then

\[ \Gamma(t_k) = \max_{P_j \in \bullet t_k} \Delta(P_j, t_k) \]

For every \( t_k \) find all places in \( \mathcal{T}_k^* \). Let \( \mathcal{P}_i \) be such a place. Compute

\[ \Delta(P_i, t_k) = \Gamma(P_i / t_k) = \tau_k + \Gamma(t_k) \]

If \( \bullet P_i \subseteq \mathcal{T}_i \) then compute

\[ \Gamma(P_i) = \max_{t_i \in \bullet P_i} \{ \Gamma(P_i / t_i) \} \]

and set \( \mathcal{P}_1 = \mathcal{P}_1 \cup \{ P_i \} \). Else set \( \mathcal{T}_1 = \mathcal{T}_1 \cup \{ t_k \} \). Repeat Step 2.

Let us now consider transformation rules which allow the construction of cost-equivalent subnets of an acyclic Ada system.

(a) Aggregation of parallel places (Figure 3a).
(b) Aggregation of parallel transitions (Figure 3b).
(c) Aggregation of serial transitions (Figure 3c).
(d) Aggregation of subnets exhibiting confusion (Figure 3d).

---

**Figure 3a.** Aggregation of parallel places. This aggregation is extended to the case when
\[ |t_1| = 1, |t_2| = n. \]

**Figure 3b.** Aggregation of parallel transitions. When \( |P_1^*| = |P_2^*| = n \) the cost associated with \( t_i \) is \( C(t'_i) = \max[\tau_1, \tau_2, \ldots, \tau_n]. \)

**Figure 3c.** Aggregation of serial transitions.

**Figure 3d.** Aggregation of subnets with serial-parallel transitions and possibly with confusion.

We conjecture that the set of transformations shown in Figure 3 allows the reduction of an acyclic Ada system to its canonical form.
As mentioned earlier the costs associated with transitions reflect the time spent by different activities in the system. In the early stages of the program development these costs will be estimates of the execution times. At a later stage these costs reflect the execution time of different activities determined at compile time using a machine dependent data base. The execution time for different instruction types, and addressing modes will be found in this data base.

A final word about Ada systems with cycles. Any analysis of such systems must be based upon the assumption that the iteration count, or the number of times a cycle is executed is either known or an upper bound of it can be estimated. In this case the cost associated with the cycle can be computed or estimated using the methodology described in this section.

4. AN EXAMPLE

Consider the Petri net in Figure 4 constructed from the Ada program in Section 2. If this net is augmented with the cost vector

$$\tau = (1, 35, 10, 5, 12, 20, 1, 9, 12, 1, 11, 1)$$

with $$\tau_1 = E(t_1)$$, $$\tau_{12} = E(t_{12})$$ and $$\tau_{i+1} = E(t_i)$$, $$1 \leq i \leq 10$$, we have an acyclic Ada system and the algorithm introduced in the previous section can be used to construct the cost matrix, \( \Delta \). Only the finite entries of \( \Delta \) are shown in Figure 5.

The incidence matrix \( N \) and the cost matrix \( \Delta \) are closely related, the non-zero entries of the incidence matrix correspond to the finite entries of the cost matrix and in Figure 5, they are represented together. The cost entry is on the right of the incidence entry.

The first few steps of the algorithm are shown below.

Step 1:

\[ \Gamma(P_{IN}) = 0 \Rightarrow \Delta(P_{IN}, t_{IN}) = 0 \]

\[ \mathcal{P}_1 = \{P_{IN}\}, \mathcal{P}_2 = P \]

\[ \mathcal{J}_1 = \emptyset, \mathcal{J}_2 = T \]

Step 2:

\[ *T_{IN} = \{P_{IN}\} \Rightarrow \Gamma(t_{IN}) = \Gamma(P_{IN}) = 0 \]

Hence \( \Delta(P_{IN}, t_{IN}) = \Gamma(t_{IN}) = 0 \). Now

\[ t_{IN}^* = \{P_1, P_2\} \]

\[ \Gamma(P_1, t_{IN}) = \tau_{IN} + \Gamma(t_{IN}) = 1 + 0 = 1 \]

\[ \Delta(P_1, t_{IN}) = \Gamma(P_1, t_{IN}) = 1 \]

But \( *P_1 = \{t_{IN}\} \Rightarrow \Gamma(P_1) = 1 \Rightarrow \mathcal{P}_1 = \{P_{IN}, P_1\} \).

\[ \Gamma(P_2, t_{IN}) = \tau_{IN} + \Gamma(t_{IN}) = 1 + 0 = 1 \]

\[ \Delta(P_2, t_{IN}) = \Gamma(P_2, t_{IN}) = 1 \]

But \( *P_2 = \{t_{IN}\} \Rightarrow \Gamma(P_2) = 1 \Rightarrow \mathcal{P}_1 = \{P_{IN}, P_1, P_2\} \).

Also \( \mathcal{J}_1 = \{t_{IN}\} \).

Repeat Step 2.

Both \( t_1 \) and \( t_2 \) satisfy the condition that all places in their presets are in \( \mathcal{P}_1 \).
$t_1 = \{P_1\} \Rightarrow \Gamma(t_1) = \Gamma(P_1) = 1$. Hence $\Delta(P_1, t_1) = 1$.

$t_2 = \{P_2\} \Rightarrow \Gamma(t_2) = \Gamma(P_2) = 1$. Hence $\Delta(P_2, t_2) = 1$.

Now

$t_1^* = \{P_{12}\}$

$\Gamma(P_{12}/t_1) = \tau_1 + \Gamma(t_1) = 35 + 1 = 36$

$\Delta(P_{12}, t_1) = \Gamma(P_{12}/t_1) = 36$

But $^*P_{12} = \{t_1\} \Rightarrow \Gamma(P_{12}) = 36$

$t_2^* = \{P_3\}$

$\Gamma(P_3/t_2) = \tau_2 + \Gamma(t_2) = 10 + 1 = 11$

$\Delta(P_3, t_2) = \Gamma(P_3/t_2) = 11$

But $^*P_3 = \{t_2\} \Rightarrow \Gamma(P_3) = 11$

It follows

$\mathcal{S}_1 = \{t_{IN}, t_1, t_2\}$

$\mathcal{P}_1 = \{P_{IN}, P_1, P_2, P_3, P_{12}\}$.

Repeat Step 2.

$t_3$ satisfies the condition that all its input places are in $\mathcal{P}_1$. $^*t_3 = \{P_3\}$. The algorithm continues until we have computed

$\Gamma(P_{12}) = 36$

$\Gamma(P_{13}) = 53$

$\Gamma(P_{14}) = 36$

and

$\mathcal{P}_2 = \{P_{OUT}\}$ and $\mathcal{S}_2 = \{t_{OUT}\}$

At this moment, all transitions in $^*t_{OUT}$ are in $\mathcal{S}_2$. It follows

$\Delta(P_{12}, t_{OUT}) = \Gamma(P_{12}) = 36$

$\Delta(P_{13}, t_{OUT}) = \Gamma(P_{13}) = 53$

$\Delta(P_{14}, t_{OUT}) = \Gamma(P_{14}) = 36$

$\Gamma(t_{OUT}) = \max_{P_m \in \mathcal{S}_{OUT}} [\Gamma(P_m)] = 53$

$\Gamma(P_{OUT}) = \tau_{OUT} + \Gamma(t_{OUT}) = 1 + 53 = 54$
Figure 4. The Petri net model of the Ada program.
<table>
<thead>
<tr>
<th>$t_{IN}$</th>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
<th>$t_4$</th>
<th>$t_5$</th>
<th>$t_6$</th>
<th>$t_7$</th>
<th>$t_8$</th>
<th>$t_9$</th>
<th>$t_{10}$</th>
<th>$t_{OUT}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Costs</td>
<td>1</td>
<td>35</td>
<td>10</td>
<td>5</td>
<td>12</td>
<td>20</td>
<td>1</td>
<td>9</td>
<td>12</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>$P_{IN}$</td>
<td>-1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_1$</td>
<td>+1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_2$</td>
<td>+1</td>
<td>1</td>
<td></td>
<td>-1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_3$</td>
<td>+1</td>
<td>11</td>
<td>-1</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_4$</td>
<td></td>
<td></td>
<td>+1</td>
<td>16</td>
<td>-1</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_5$</td>
<td></td>
<td></td>
<td>+1</td>
<td>16</td>
<td>+1</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_6$</td>
<td></td>
<td></td>
<td>+1</td>
<td>28</td>
<td>-1</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_7$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+1</td>
<td>29</td>
<td>-1</td>
<td>29</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_8$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+1</td>
<td>29</td>
<td></td>
<td>-1</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_9$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+1</td>
<td>30</td>
<td></td>
<td>-1</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_{10}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+1</td>
<td>41</td>
<td></td>
<td>-1</td>
<td>41</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P_{11}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+1</td>
<td>42</td>
<td>-1</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>$P_{12}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+1</td>
<td>36</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-1 36</td>
</tr>
<tr>
<td>$P_{13}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+1</td>
<td>53</td>
<td>-1 53</td>
<td></td>
</tr>
<tr>
<td>$P_{14}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+1</td>
<td>36</td>
<td></td>
<td>-1 36</td>
</tr>
<tr>
<td>$P_{OUT}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+1 54</td>
</tr>
</tbody>
</table>

**Figure 5.** The incidence and the cost matrices of the Ada system in Figure 4.
BIBLIOGRAPHY


