

Project Management Using Point Graphs¹

Abbas K. Zaidi², Mashhood Ishaque, Alexander H. Levis

System Architectures Laboratory, ECE Dept., MS 1G5
George Mason University, Fairfax, VA 22030-4444 USA
<szaidi2>, <alevis>@gmu.edu

Abstract

The paper presents a graph-based approach for scheduling and monitoring temporal events and activities in a system engineering project. It attempts to overcome some of the limitations of traditional project management approaches by allowing specifications of real-time *milestones*, and by breaking the *finish-start* barrier between the activities. An illustrative example presents the approach.

Introduction

The traditional project management and concurrent engineering approaches date back to 1950's with the advent of the Critical Path Method (CPM), developed as a result of a joint venture between the DuPont and Remington Rand corporations to study the trade-offs between the cost of a project and its overall completion time. In 1958, another approach called Project Evaluation and Review Technique (PERT) was invented by Booz Allen Hamilton, Inc. under contract to DoD's US Navy Special Projects Office as part of the Polaris missile project with an emphasis on shortening the overall completion time of the system under development. (Moder and Philips, 1970) The two approaches, commonly

called CPM/PERT, provide an analytic underpinning to product/project management/monitoring problems by addressing shortest completion time, cost-time tradeoffs, and scheduling issues (i.e., critical activities and time slacks) for both deterministic and probabilistic concurrent activities. These are essentially paper-and-pencil techniques that have been implemented as (or in several) software tools by many organizations and individuals, and have been extensively used by professionals since their inception. The approaches have certain limitation in terms of the type of activities and the types of temporal constraints allowed between activities. For example, both are limited to only duration based activities, and the only temporal constraint allowed is between the end of an activity and the start of a following activity; this is referred to as the finish-start barrier. This results in specification of sequential and totally parallel activities, but with no provision of requirements for *nominally* parallel tasks/activities. Since their inception, there has been tremendous work done on scheduling problems with a lot of emphasis on resource allocation for different types of resources required for project activities; however, we find very little effort on overcoming the stated shortcomings of the traditional project management approaches. The size, complexity,

¹ The work was carried out with support provided by the Air Force Office of Scientific Research under contract number FA9550-05-1-0106.

² Corresponding author

and distributed nature of modern-day systems (or systems of systems) and the engineering processes required to design such systems call for a robust set of tools supporting different aspects of the system engineering and management processes. The approach presented in this paper contributes to this tool suite by overcoming some of the weaknesses in the traditional approaches to project management and monitoring.

This paper proposes a graph-based representation, called Point Graphs (PG), to model a class of temporal relations between points and/or intervals. The points and intervals represent time stamps and time delays associated with events and activities in a project plan, respectively. The graph representation is shown to enable analyses similar to those of classical scheduling approaches. The graph representation is supported by an input specification language for project activities and temporal constraints among them. The language and the graph representation used in the approach are more expressive than the project networks used for the traditional approaches. The graph structure of the PG is also used to reveal and identify corrections for inconsistencies and temporal anomalies, if present in the input.

The Point Graphs (PG) and the accompanying *Point-Interval* formalism (PIL) originated from an earlier work (Zaidi, 1999, 2001; Zaidi and Levis, 2001; Ishaque and Zaidi, 2005; Zaidi and Wagenhals, 2006) on temporal knowledge representation and reasoning. The work in this paper extends the earlier work by:

1. Developing a suite of algorithms, based on the graphical representation of Point Graphs (PG) that determine the shortest duration for the entire set of activities, earliest/latest start/end times of temporal activities, critical activities, and time slacks for non-critical activities.
2. Enhancing the classical approaches to temporal planning, e.g., CPM and PERT, by in-

corporating the provision for *nominally* parallel relations among intervals. For example, a temporal constraint of the type “Activity X starts before Activity Y ends” (formally, ‘ $sX \leq eY$ ’, where $X = [sX, eX]$ and $Y = [sY, eY]$) cannot be handled by the classical approaches.

3. Allowing the provisions for exact deadlines, i.e., milestones, for start/end of activities, specification of ‘at least’ and ‘at most’ type constraints on start/end/completion times of activities, etc. This feature is especially useful for real-time monitoring of a project’s progress with activities either falling behind the schedule and/or being completed earlier than scheduled. The results of the analyses can help project managers re-adjust their project schedules/plans with real-time feedback.
4. Implementing the approach in a software application that incorporates a hierarchical Point Graph representation to support both top-down and bottom-up project management paradigms, which is especially useful for large-scale systems.

Point Graph

Figure 1 presents a graph construct called Point Graphs (PG) that is used to model temporal information in a project. In a PG, a node represents a point (or a *composite* point) and an edge between two points represents one of the two temporal relations, *before* and *precedes*, between the two. An interval in this graph representation is depicted as a pair of start and end points with a LT edge connecting the two. A formal definition of Point Graphs is given as follows:

Definition: Point Graphs

A Point Graph, PG (V, E_A, D, T) is a directed graph with:

V : Set of vertices with each node or vertex $v \in V$ representing a point on the real number line. Points p_i, p_j, \dots, p_n are represented as a

composite point $[p_i; p_j; \dots; p_n]$, if all are mapped to a single point on the line.

E_A : Union of two sets of edges: $E_A = E \cup E_{\leq}$, where

E : Set of edges with each edge $e_{12} \in E$, between two vertices v_1 and v_2 , also denoted as (v_1, v_2) , representing a relation ' $<$ ' (*before*) between the two vertices—($v_1 < v_2$). The edges in this set are called LT edges;

E_{\leq} : Set of edges with each edge $e_{12} \in E_{\leq}$, between two vertices v_1 and v_2 , also denoted as (v_1, v_2) , representing a relation ' \leq ' (*precedes*) between the two vertices—($v_1 \leq v_2$). The edges in this set are called LE edges.

D: Edge-length function (assumed total): $E \rightarrow \mathbb{R}^+$

T: Vertex-stamp function (possibly partial): $V \rightarrow \mathbb{R}$

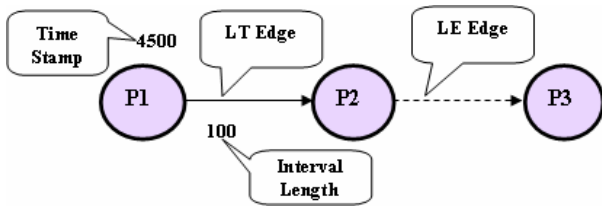


Figure 1. Point Graph Construct.

Figures 2 & 3 illustrate how this PG representation can be related to the constructs of an input language. The language, called PIL, can be used to specify temporal constraints, both qualitative and quantitative, among intervals and events representing tasks and milestones in a project. The temporal constraints in Fig. 3 require introduction of virtual time point(s) or virtual nodes in a PG. A virtual node is like any other node in a PG except for the fact that there is no temporal variable (point, start of interval, or end of interval) associated with it. Therefore, it does not have a unique identifier or *name* associated with it.

Point P, Q	
P Before Q	
P Equals Q	
P Precedes Q	

(a) Point to Point (PP) Constraints

Point P, Interval Y	
P Before Y	
P Starts Y	
P During Y	
P Finishes Y	
Y Before P	

(b) Point to Interval (PI) Constraints

Interval X, Y	
X Before Y	
X Precedes Y	
X Meets Y	
X Starts Y	
X During Y	
X Finishes Y	
X Equals Y	

(c) Interval to Interval (II) Constraints

Figure 2. Specification Language and PG Representation.

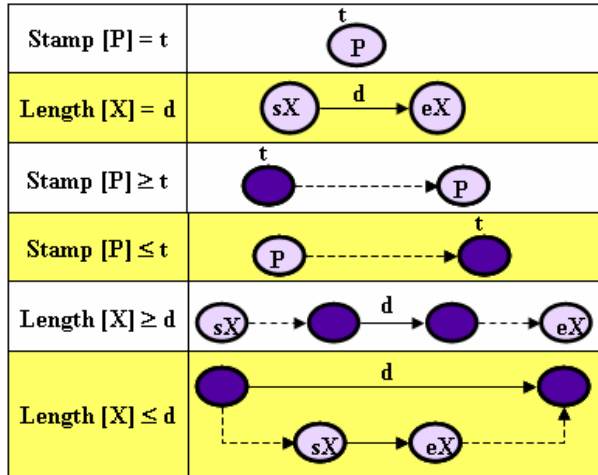


Figure 3. Quantitative Constraints and their PG Representations.

Operations on Point Graphs. The approach in this paper starts with a set of PIL statements representing the temporal entities and the constraints to be satisfied. Each statement in this set can be converted to an equivalent graphical representation using the mapping shown in Figs. 2-3. The PG representing the entire set of statements is then constructed by *unifying* individual PGs to a (possibly) single connected graph. The unifying process looks at the labels of the nodes (except for virtual nodes) and the values of the time stamps associated with them to identify equalities. The nodes identified as being equal to one another are merged into a single node with a composite label. The unified PG is then *folded* with the help of lengths on edges. This folding process establishes new relations among system intervals, inferred through the quantitative analysis of the known relations specified by interval lengths and stamps. Fig. 4 illustrates the two operations on an example set of PGs. The input constraints may be infeasible; therefore, the resulting PG is checked for inconsistency, as illustrated in Figure 4c. The infeasibility in the input constraints reveals itself in a PG either in the form of cycles or with the presence of multiple paths between a pair of nodes with conflicting path lengths. A more technical and detailed description of these graph operations, their

computational complexity, and the verification mechanism can be found in Zaidi and Wagenhals (2006), and Ishaque (2006). Once a PG is verified, a couple of virtual nodes are added to it as illustrated in Fig. 4d. A virtual source (sink) node V_{in} (V_{out}) is added to PG with LE arcs connecting V_{in} (V_{out}) to all the source (sink) nodes in the PG as shown in the figure. The two nodes represent the overall start-to-end time for the project under consideration. Once a PG for the project is constructed with the help of steps illustrated in Fig. 4, a set of algorithms, presented in the following section, is applied to it.

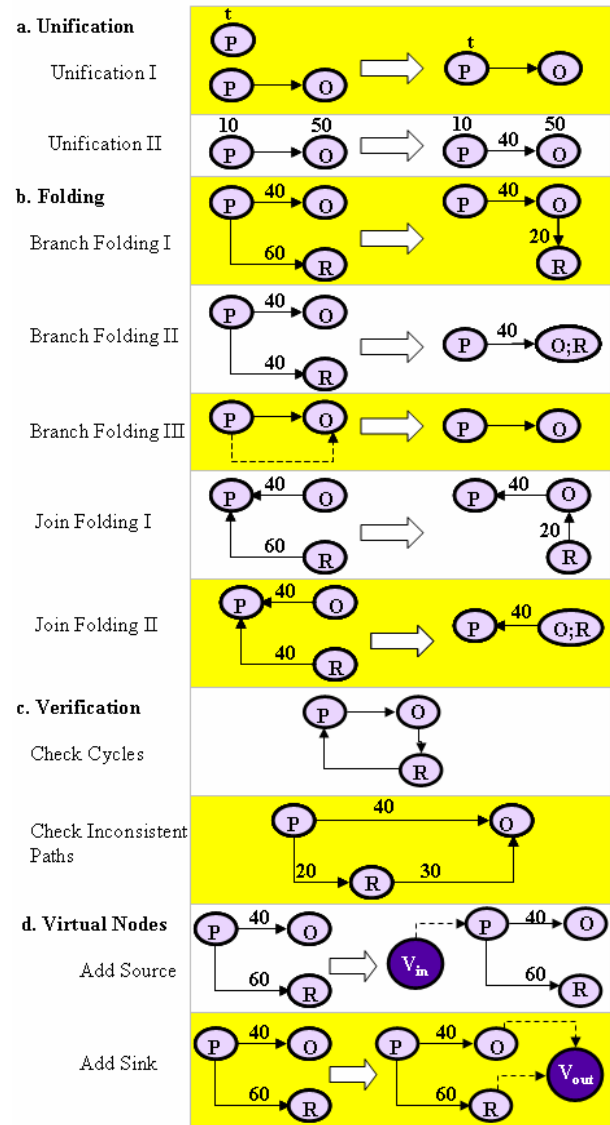


Figure 4. Operations on a Point Graph.

Scheduling Algorithms

The scheduling algorithms applied to the PG representation calculate three parameters for each node in the PG. The parameter values are calculated by running two sets of algorithms, *Forward** followed by *Reverse**, on the graph. The values of these parameters help determine the *critical activities* and time floats/slacks for intervals in the system, and *interval/point activities* defined for the PG under consideration. The three parameters are called *earliest occurrence* (E_v), *late occurrence* (L_v), and *latest occurrence* (T_v) of a node 'v', and are defined as follows.

Earliest Occurrence, E_v . The earliest occurrence E_v of a node v is the smallest time stamp on the node that satisfies the earliest occurrences of the preceding nodes. Figure 7 illustrates the method of calculating a node's earliest occurrence time with the help of earliest times on its child (preceding) nodes. The manner in which this parameter is calculated for all the nodes in a PG requires a *forward* traversal of the PG starting from the sink node, which by default is given a 0 value for the earliest occurrence time.

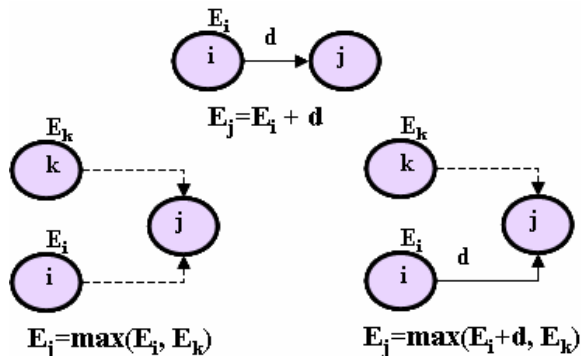


Figure 7. Earliest Occurrence Time.

Late (Latest) Occurrence, L_v (T_v). The late (latest) occurrence L_v (T_v) of a node v is the largest time stamp on the node that satisfies the earliest (latest) occurrences of the following nodes. Fig. 8 illustrates the method of calculating a node's late and latest occurrence times with the help of corresponding parameter values on its parent (fol-

lowing) nodes. The calculation for these two parameters requires a *reverse* traversal of the PG starting from the sink node, which is by default initialized to the earliest occurrence time, calculated during the forward sweep, for both late and latest occurrence times.

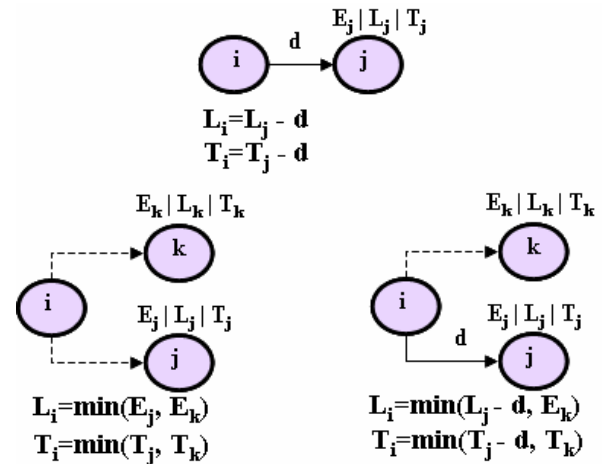


Figure 8. Late & Latest Occurrence Times.

Notice when a stamp (i.e., as a deadline or milestone) has been provided for a node in the Point Graph, the stamp represents a hard constraint on the occurrence time of the node. Thus the earliest, late and latest occurrence times must be equal to the stamp of the node; the calculation illustrated in Figs. 7 and 8 are adjusted accordingly. The two passes on the PG representation are now presented in more detail in the following sections.

Forward*. The forward pass comprises two algorithms, as described in Tables 1 & 2, that calculate E_v for each node v in the PG. The first algorithm, the Forward I (Table 1) simply traverses the graph and calculates the parameter values as illustrated in Figure 7. The time complexity of the algorithm is $O(m+n)$ where m is the number of edges and n is the number of nodes in a PG. After the application of this algorithm, it is possible for an activity to have *Stretch Float*, i.e., the earliest start of the activity does not ensure its earliest completion; or the activity needs to be stretched beyond its duration in order to complete the project on time. The concept was first introduced in Zaidi and Wagenhals

(2006). A real life example of stretch float can be found in air transportation. A plane takes off at location A and lands at location B, the time difference between the landing and take off may be greater than the minimum travel time between the two destinations. In case an activity is not stretchable, the earliest occurrence time needs to be recalculated in the PG. Fig. 9 shows the PG in Example 1 (Fig. 5) after it was processed by the Forward I algorithm. Activity D is shown with a stretch float equal to 3.

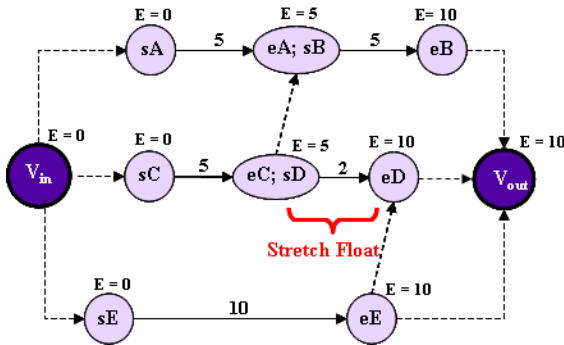


Figure 9. A PG with Stretch Float

The second algorithm, called Forward II (Table 2) re-calculates the earliest occurrence time for each activity without the stretch float. The algorithm is not needed if the application of Forward I results in no reported stretch floats or if the reported stretch floats are associated with stretchable activities. The algorithm, when applied, works by *fixing* the edges where stretch float occurs. Fixing an edge might result in a situation where other edges (both LT and LE) may need to be fixed as well. But the algorithm always fixes at least one edge in every iteration; a maximum of $|E|$ iterations will fix all edges. In each iteration, the algorithm examines all edges. So the time complexity of the algorithm is $O(m^2)$. The two algorithms are collectively called Forward*.

Reverse*. The application of Forward* is followed by a similar set of algorithms, collectively called Reverse*, for the calculation of late and latest occurrence times for every node in the PG. Tables 3 and 4 describe the two algorithms for late occurrence time. The calculation of latest follows the steps in Tables

3-4; the only difference is in the formula used to calculate the parameter values.

Fig. 10 shows the PG of Example 1 after it is processed by both Forward* and Reverse* algorithms with no provision for stretch float which required the application of the Forward II and Reverse II algorithms on it.

Table 1. The Forward I Algorithm.

```

Calculating  $E_v$  for node  $v$ 
(starting from  $V_{out}$ )
Let  $*v$  be the pre-set of  $v$ 
Check for node stamp:
IF  $v$  has stamp
    Set  $E_v = \text{stamp}[v]$ 
ELSE
    Set  $E_v = 0$  //initialization
FOR each node  $v_i$  in  $*v$ 
    Calculate  $E_{v_i}$  recursively.
    IF  $v$  does not have stamp
        IF  $(v_i, v)$  is LT edge
             $E_v = \max(E_v, E_{v_i} + \text{Length}(v_i, v))$ 
        ELSE IF  $(v_i, v)$  is LE edge
             $E_v = \max(E_v, E_{v_i})$ 

```

Table 2. The Forward II Algorithm.

```

FOR  $|E|$  times
(where  $E$  is set of edges in the PG)
    Set Modified = false.
    FOR each edge  $(v_i, v_j)$ 
        IF  $(v_i, v_j)$  is a ' $\leq$ ' edge
            IF  $E_i > E_j$ 
                Set  $E_j = E_i$ . Set Modified = true.
        ELSE
            IF  $E_j > E_i + \text{Length}(v_i, v_j)$ 
                Set  $E_i = E_j - \text{Length}(v_i, v_j)$ .
                Set Modified = true.
            ELSE IF  $E_j < E_i + \text{Length}(v_i, v_j)$ 
                Set  $E_j = E_i + \text{Length}(v_i, v_j)$ .
                Set Modified = true.
    IF (Modified = false)
        Return

```

Table 3. The Reverse I Algorithm.

```

Calculate  $L_v$  for node  $v$  (starting from  $V_{in}$ )
Let  $v^*$  be the post-set of  $v$ 
Set  $L_v = E_v$  //initialization
FOR each node  $v_i$  in  $v^*$ 
    Calculate  $L_{v_i}$  recursively
    IF  $v$  does not have stamp
        IF  $(v, v_i)$  is LT edge
             $L_v = \min(L_v, L_{v_i} - \text{Length}(v, v_i))$ 
        ELSE IF  $(v, v_i)$  is ' $\leq$ ' edge
             $L_v = \min(L_v, E_{v_i})$ 
    
```

Table 4. The Reverse II Algorithm.

```

FOR  $|E|$  times (where  $E$  is the set of edges)
Set Modified = false.
FOR each edge  $(v_i, v_j)$ 
    IF  $(v_i, v_j)$  is a ' $<$ ' edge
        IF  $L_j > L_i + D(v_i, v_j)$ 
            Set  $L_j = L_i + D(v_i, v_j)$ .
            Set Modified = true.
        ELSE IF  $L_j < L_i + D(v_i, v_j)$ 
            Set  $L_i = L_j - D(v_i, v_j)$ .
            Set Modified = true.
    IF (Modified = false)
Return
/*The algorithm for latest occurrence
replaces  $L$  parameter with  $T$ .*/
    
```

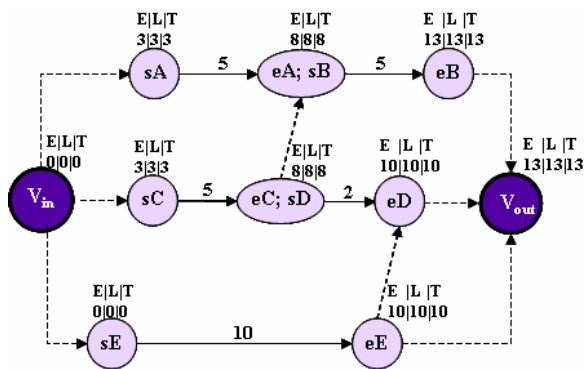


Figure 10. PG of Example 1 after Processing.

Finally the PG corresponding to a project's requirements with the values of the parameters calculated is used to construct a time chart, e.g. Gantt chart, showing the start

and finish times for each activity as well as its relationship to other activities. The parameters values are used to identify *critical* and non-critical activities in the project. For non-critical activities the amount of slacks or *floats* are calculated that can be used advantageously when such activities are delayed or when limited resources are to be used. The following describes the notion of a critical activity and different types of floats that are calculated after the application of graph algorithms.

Critical Activity. An activity is defined to be critical if a delay in its start will cause a delay in the completion time of the entire project, i.e.,

- i. For a point activity $v \in V$, $E_v = T_v$; for an interval activity $[v_1, v_2]$, where $v_1, v_2 \in V$, $v \in \{v_1, v_2\}$, $E_v = T_v$, or
- ii. For an interval activity, it 'Meets' or is met by another critical activity; for a point activity, it 'Starts' and/or 'Ends' another critical activity.

Total Float (TF) and Free Float (FF). Total Float (TF) is the difference between the maximum time available to perform an activity and its duration. Free Float (FF) for an activity is defined by assuming that all the activities start as early as possible, it is the excess time available over its duration.

(a) Total float (TF) and free float (FF) for a non-critical point activity v are calculated as: $TF_v = T_v - E_v$

$$FF_v = L_v - E_v$$

(b) Total float (TF) and free float (FF) for a non-critical interval activity $[v_1, v_2]$ are calculated as:

$$TF_{[v_1, v_2]} = T_{v_2} - E_{v_2} = T_{v_1} - E_{v_1}$$

$$FF_{[v_1, v_2]} = L_{v_2} - E_{v_2} = L_{v_1} - E_{v_1}$$

(For all critical activities $TF = FF = 0$.)

Table 5 shows these calculated values for the system in Example 1.

Temper

The approach presented in the last section has been implemented as a software tool

called Temper (Temporal Programmer). Temper provides a language editor to input PIL statements. It has a graphical interface to display the Point Graphs and also a text I/O interface to display information and results of the analyses. The algorithms are implemented in the form of a .NET class library called *PIL Engine* that provides an application programming interface (API) which can be used in any .NET compliant programming language. Temper has been built using this API. It provides a graphical user interface to *PIL Engine*. It uses *QuickGraph*, which is an open-source C# implementation of *Boost Graph Library* (BGL), and *Graphviz* library from AT&T (Graphviz), for internal graph representation and for implementation of PIL algorithms.

Table 5. Parameter Values for Activities in Example 1.

Activity	A	B	C	D	E
Duration	5	5	5	2	10
Earliest Start Time	3	8	3	8	0
Latest End Time	8	13	8	10	10
Critical	Yes	Yes	Yes	Yes	Yes
Total Float	0	0	0	0	0
Free Float	0	0	0	0	0

Since all the algorithms presented have polynomial time complexity, the software implementation not only can handle large scale system engineering projects but can also be used for a real-time monitoring of the activities. The allowance for exact stamps or for specifying bounds on stamps for start/end of activities in the approach can be used to add additional constraints (especially during project execution) reflecting actual and more accurate times/dates for the activities involved. With the new information added, better estimates for completion times and time slacks can be re-calculated to determine if things are falling behind the schedule and/or being completed earlier than scheduled. The results of the analyses can help project managers re-adjust their project schedules/plans/cost esti-

mates with real-time feedback. The following example illustrates the approach.

Example 2. Suppose in the system of Example 1, the new information requires that Activity E can start no earlier than time 3, and Activity B must start at time 12. The two temporal constraints are captured by the following PIL statements that are added to the PIL statements in Example 1.

$$\text{Stamp [sE]} \geq 3 \quad \text{and} \quad \text{Stamp [sB]} = 12$$

Fig. 11 shows the new PG constructed for the new input with the new parameter values calculated for each node. The figure does not show the virtual nodes and is a simplified version of the actual PG that will be generated for the input. Table 6 shows the new parameter values for the activities involved in the project.

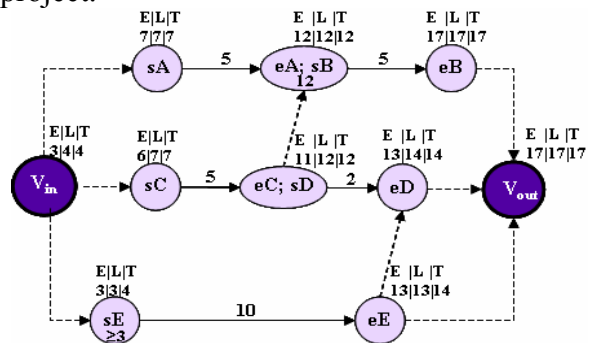


Figure 11. PG for Example 2.

Hierarchical Point Graphs. Temper also has a provision for planning and managing project activities using Hierarchical Point Graphs (HPG). This feature enables a manager to work at a higher level of abstraction but still generate feasible schedules consistent with the low-level system details. HPGs provide the capability to combine a high-level plan with the detailed sub-plans. Fig. 12 illustrates this hierarchical arrangement with the help of the PG from Example 1 in which the activity E is shown substituted by a detailed PG with the low-level activities and constraints. This encourages a modular and distributive approach to project management in which detailed plans can be developed separately from the high-level tasks representing them with completion times used as constraints that bind

different levels together. This hierarchical manner of constructing project plans can be done using both bottom-up and top-down approaches. In the bottom up approach, completion times for sub-plans can be added as constraints for the abstract high-level activities representing them. In the top-down approach, the earliest start and latest completion times calculated for a high-level activity become constraints for the low-level sub-plan substituting the activity.

Table 6. Example 2 Parameter Values.

Activity	A	B	C	D	E
Duration	5	5	5	2	10
Earliest Start Time	7	12	6	11	3
Latest End Time	12	17	12	14	14
Critical	Yes	Yes	No	No	No
Total Float	0	0	1	1	1
Free Float	0	0	1	1	0

Conclusion

The approach presented in this paper extends the classical duration-based quantitative approaches for project management and monitoring by adding the provision for point (instantaneous) activities and specification of partially ordered relation between system activities. It also offers an expressive input language for project managers to input their specifications. The accompanying software offers an analytic toolkit for project/system managers and engineers for planning and monitoring large scale development projects as well as for scheduling services in the modern-day system-of-systems that acquire and lose services or parts of other systems at run-time to develop new and unprecedented capabilities.

Biography

Dr. Abbas K. Zaidi is Research Associate Professor with the System Architectures Laboratory, ECE Dept., George Mason University.

Dr. Alexander H. Levis is University Professor of Electrical, Computer and Systems Engineering at George Mason University.

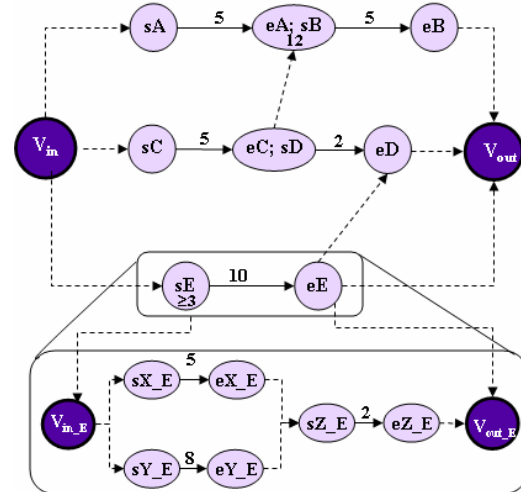


Figure 12. A Hierarchical PG

References

- Boost Graph Library: <http://www.boost.org/>
- Graphviz: <http://www.graphviz.org/>
- Ishaque, M and A. K. Zaidi, "Time-Sensitive Planning Using Point-Interval Logic," *10th International Command and Control Research and Technology Symposium*, 2005.
- Ishaque, S. M. M. "On Temporal Planning and Reasoning with Point Interval Logic," *MS Thesis*, CS, George Mason University, VA.
- Moder, J., and C. Philips, *Project Management with CPM and PERT*, 2nd Edition, Van Nostrand Reinhold, NY, 1970.
- Zaidi, A. K. and A. H. Levis, "TEMPER: A Temporal Programmer for Time-sensitive Control of Discrete-event Systems," *IEEE Transaction on SMC*, 31, 6, 485-496, 2001.
- Zaidi, A. K. and Lee W. Wagenhals, "Planning Temporal Events Using Point Interval Logic," *Special Issue of Mathematical and Computer Modeling*, vol 43, Elsevier, 2006.
- Zaidi, A. K., "A Temporal Programmer for Time-Sensitive Modeling of Discrete-Event Systems," in *Proc. of IEEE - SMC 2000 Meeting*, Nashville, TN, October 2001.
- Zaidi, A. K., "On Temporal Logic Programming Using Petri Nets," *IEEE Transactions on SMC*, Part A, 29(3): 245-254, May 1999.

Mashhood Ishaque worked as a Graduate Research Assistant in the System Architectures Lab of George Mason University. He received his Masters of Science in Computer Science from George Mason University in 2006. Currently he is a doctoral student in Computer Science at Tufts University.