JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 16, 423-446 (2000)

Short Paper

A Project Model for Software Development

BIN-SHIANG LIANG, JENN-NAN CHEN* AND FENG-JIAN WANG
Institute of Computer Science and Information Engineering
National Chiao Tung University
Hsinchu, Taiwan 300, R.O.C.
E-mail: fjwang@csie.nctu.edu.tw
**Samar Techtronics Cooperation Ltd.*
Taiwan, R.O.C.

Uncertainty and dynamic changes in a software project cause iterations during development and the need for decision-making in planning and controlling a project. This paper presents a *Software Project Review and Evaluation Model, SPREM*, a superset of *CPM/PERT,* which extends *CPM/PERT'*s notation to four types of vertices (*AA*, *AX*, *XA*, and *XX* vertices) to express the non-deterministic and iterative behaviors of software engineering projects. Several behavioral properties of *SPREM* and analysis of them are discussed. For example, the enaction capability can be used to evaluate the possibility that a vertex will enact processes beforehand. Project managers can revise a *SPREM* graph to rescue dead vertices before project execution. Furthermore, enaction ordering allows project managers to calculate the dependency between processes to be enacted. This might help in computing important information such as critical paths among these processes.

***Keywords*:** CPM/PERT, software project management, SPREM, software process modeling, project evaluation

1. INTRODUCTION

The CPM (Critical Path Method)/PERT (Plan Evaluation and Review Technique) network has been applied broadly to project management. It is an acyclic and directed AOV (Activity On Vertex)[1] network able to express sequential, parallel, and synchronous activity behaviors [3, 20]. However, its deterministic[2] and acyclic properties are not suitable for managing projects consisting of iteratively and decisionally-based procedures [20]. Especially, it is not suitable for software development, due to the lack of iteration and decision-making [5, 7, 20, 23]. Many studies have been devoted to extending CPM/PERT to modeling of the behaviors of projects with a high degree of uncertainty and dynamically changing

Received September 15, 1997; revised May 14, 1998; accepted August 20, 1998.
Communicated by Jieh Hsiang.

[1] A CPM/PERT can be either an AOV or an equivalent AOA (Activity On Arc) network. Here, AOV is adapted for comparison with SPREM.

[2] Here, deterministic means that when a process on vertex V is complete, then all of V's successors will enact the process individually.

characteristics. These studies can be divided into two categories: one category extends the different types of CPM/PERT network vertices [9, 10, 12, 27], and the other combines CPM/PERT with Petri nets [20, 21, 25]. GANs (Generalized Activities Networks) [12] introduces a simple dichotomous choice at each vertex in order to provide a 'decision-box planning and scheduling' technique for research projects.

GAN, devised in [10], views vertices from input and output logic perspectives. Unfortunately, these papers seem unclear in their definition [9]. GERT (Graphical Evaluation and Review Technique) is based on ideas introduced in [27] and adds notations concerning Exclusive-OR (XOR) vertices and the probability of OR/XOR vertices existing in the network. Clearly, this method is restricted by analysis limitations, and its analysis relies only on simulations. Design/Net [20] combines AND-OR graphs with Petri net notation to describe and monitor software processes. However, it is acyclic, and little analysis is provided. [25] combined Beat-distributed stochastic Petri nets with PERT, and [21] combined generalized stochastic Petri nets with PERT for performance evaluation of concurrent processes. Both have problems with computational complexity in exploring state spaces during analysis, and currently must restrict nets to being acyclic to reduce the computational complexity. In summary, the former category is limited to reliance on simulation [9]. As for the latter, evaluating the performance of Petri nets even with deterministic timing and firing frequencies per cycle for all state transitions is often an NP-hard problem [29]. Furthermore, this approach seems unsuitable for managing large and complicated projects because Petri nets use lower-level semantics for comprehension.

This paper presents the *Software Project Evaluation and Review Model* (SPREM), a superset of CPM/PERT that extends CPM/PERT's notation to four types of vertices (*AA*, *AX*, *XA*, and *XX* vertices). SPREM can express the various behaviors required to model software engineering projects. A$*$ (AX and AX) vertices are able to model the synchronization of a process while $*$A (AA and XA) vertices can model the fork of parallel processes. X$*$ (XA and XX) vertices are able to model what-if processes while $*$X (AX and XX) vertices model decision-making processes. Looping is allowed in SPREM so as to model iterative processes. With SPREM, managers can model large and complicated projects using top-down and/or bottom-up approaches since a vertex can be decomposed into a SPREM graph containing a set of vertices. Also, analysis in SPREM, *such as* enaction capability and performance analysis, can be computed with a reasonable cost. Enaction capability analysis can be used to evaluate the potential of a vertex to enact processes beforehand. Project managers can revise a SPREM graph to rescue dead vertices before project execution. In addition, enaction ordering allows project managers to calculate the dependency between the processes to be enacted. This might help in computing important information, such as critical paths among these processes.

The rest of this paper is organized as follows. Section 2 defines SPREM and discusses its enaction behaviors. To demonstrate the modeling ability of SPREM, section 3 presents an example based on the spiral model paradigm [4], widely accepted as a realistic software development paradigm. The example indicates that with SPREM, it is easier to model a software project than it is with a Petri net. In section 4, some interesting characteristics of SPREM are defined and discussed. Sections 5 and 6 present the analysis algorithms of SPREM, which are of interest during project planning and are introduced at two levels, *acyclic* and *cyclic*, respectively. Some conclusions and future work are described in section 7.

2. A SPREM GRAPH

2.1 Definition of SPREM

A SPREM graph is a Five-tuple <**V**, **A**, $\boldsymbol{S_p}$, $\boldsymbol{E_p}$, $\boldsymbol{M_0}$>. **V** = {V_1, V_2, ..., V_m} is a finite set of vertices, where m, $m \geq 0$, is the number of vertices. There are four *types* of vertices: *AA* (*AND-AND*), *AX* (*AND-XOR*), *XA* (*XOR-AND*), and *XX* (*XOR-XOR*), where *A* represents "all", *X* represents "exclusion", and the first character represents the condition that enacts the vertex process while the second represents what is to be done at the end of the process. Fig. 1 (a-d) shows the notation associated with AA, AX, XA, and XX vertices, respectively.

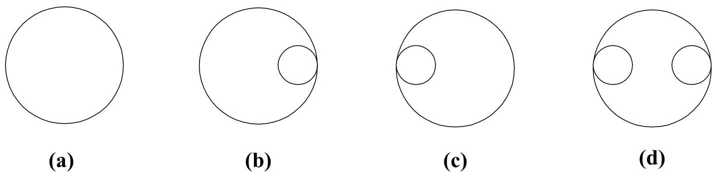

Fig. 1. The notation associated with four types of vertices in SPREM.

A: $\mathbf{V} \times \mathbf{V}$ = {A_1, A_2, ..., A_n} is a finite set of directed arcs, where n, $n \geq 0$, is the number of arcs. Each arc is associated with a Boolean value. Each pair of vertices contains at most one directed arc. If a vertex contains only an input arc, the vertex is both an *A*∗ vertex and an *X*∗ vertex. When a SPREM graph contains only AA vertices and no loops, the graph is a CPM/PERT network.

$\boldsymbol{S_p}$, $\boldsymbol{E_p}$**:** There is only one entry vertex called *Start Project*, $\boldsymbol{S_p}$, and only one exit vertex called *End Project*, $\boldsymbol{E_p}$, in any SPREM graph. To simplify discussion, $\boldsymbol{S_p}$ has no input, and $\boldsymbol{E_p}$ has no output arcs.

$\boldsymbol{M_0}$**:** $\mathbf{A} \rightarrow \mathbf{B}$ is an initial SPREM graph indicating where **B** is a set of Boolean values.

A vertex is *enactable* if its precondition holds; i.e., the value of its input arc(s) satisfies an "AND" or "XOR" condition. An A∗ vertex is enactable when all its input arcs are set to TRUE. An X∗ vertex is enactable when one of its input arcs is set to TRUE and the rest are FALSE.

The Boolean values of arcs are assigned initially and then set or reset by its tail or head during execution. The precondition of all vertices cannot hold at the beginning; i.e., only vertex $\boldsymbol{S_p}$ can enact a process at the beginning.

A vertex enacts a process once, including instantiating, running, and then terminating the process. To simplify discussion, we assume that there is no condition checking, arc value changing, or time consumption for process instantiation/termination. Every input arc of an enabled vertex is set to FALSE while the vertex is instantiating a process. A process starts to run as soon as it is instantiated. A vertex is *active* when it has a process running. An enactable vertex cannot enact a new process if it is active; *that is*, a vertex cannot enact a new process when it has a process running even though value changes in its input arc(s) make it enactable. After the running process has been terminated, the vertex enacts a new process if it is enactable. The values of the vertex output arcs can be treated as contributions for the termination of processes associated with the vertex. When ∗A vertex V terminates its process, it sets all V's output arcs to TRUE. When ∗X vertex V terminates its process, it sets one of V's output arcs to TRUE and the rest to FALSE, no matter what their values were previously.

A vertex in a SPREM graph can represent another SPREM graph. For vertex V_a representing SPREM graph SP_a, instantiating for V_a means instantiating $\boldsymbol{S_p}$ in SP_a. If the process in V_a runs, this means that the vertices in SP_a except for $\boldsymbol{S_p}$, run processes. The design of SP_a is erroneous, if $\boldsymbol{E_p}$ cannot enact a process. $\boldsymbol{E_p}$ in SP_a may enact more than one process. SP_a is considered complete only when no running processes exist in any vertices of SP_a. The process in V_a is terminated when SP_a finishes (due to enaction).

Obviously, a SPREM graph may allow a vertex to enact more than one process even though it contains no loop. For example, let X∗ vertex V contain only two predecessors, which are enacting two distinct processes p_a and p_b, where p_a terminates earlier than p_b. Assume that the process enacted by V terminates between the termination of p_a and p_b. V will enact one new process when process p_b is terminated. In other words, V enacts two processes.

3. AN EXAMPLE OF MODELING A SPIRAL MODEL

To demonstrate the modeling ability of SPREM, an example based on the spiral model paradigm [4] is presented in this section. This paradigm for software engineering is one realistic approach to the development of large-scale software systems [26]. The paradigm defines four major phases to be iterated evolutionarily during each development cycle: (1) *planning* - determination of objectives, alternatives, and constraints, (2) *risk analysis* – analysis of alternatives and identification/resolution of risks, (3) *engineering* – development of the "next-level" product, and (4) *customer evaluation* – assessment of the results of engineering. Obviously, the CPM/PERT technique fails to model this paradigm naturally.

3.1 A SPREM Example

When a development process is modeled using SPREM, its referred/generated artifacts can be modeled as the attributes associated with the input/output arcs of a vertex. The artifacts must remain at some state and are changed after a process is enacted. Also, information and/or states of resources and staffs can be modeled as vertex attributes to express the requirements needed to enact processes. Fig. 2 shows the paradigm modeled using SPREM. The processes/artifacts along each vertex/arc in Fig. 2 are listed in (Appendix) Table 5 and Table 6, respectively. Fig. 10 in the Appendix shows the state transition diagram of a software artifact manipulated by processes.

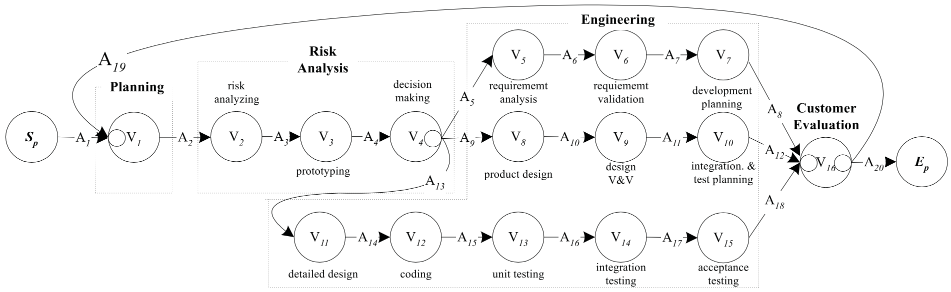

Fig. 2. An example of a spiral model created by SPREM.

In Fig. 2, the paradigm is modeled as an iterative cycle identified as A_{19} (the LBA) incident from V_{16} to V_1 and containing four phases. Each phase is shown within the region enclosed by a dashed line. The *planning* phase contains V_1 only to express the process *determine-objectives-alternatives-and-constraints*. Assume that the project is required to produce a product called artifact *software*. *Software* stays in an *initialized* state at the beginning, described as A_1. After the *planning* phase *software* is translated into a *planned* state, described as A_2, the project enters the *risk-analysis* phase, which contains V_2, V_3, and V_4. V_2 enacts the *risk-analysis* process to analyze alternatives and identify/resolve risks. After risks have been analyzed, described as A_3, a prototype of *software* is produced, described as A_4, by the process *prototyping* enacted by V_3.

Each time a prototype is generated and simulated (or measured), the *decision-making* process on V_4 (an AX vertex) will choose one thread in the *engineering* phase to enact processes. For example, enacting process *product-design* on vertex V_8 requires that artifact *product-design-specification* be *initialized* or *modified*, *i.e.*, requires A_9 to be TRUE. The *Engineering* phase contains three threads: requirement analysis, product design, and detailed design. Each time one thread is chosen and executed, a process in the *customer-evaluation* phase is followed to plan the "next-level" product and assess the results of engineering *software*. For example, if the product design phase is chosen, then the process *product-design* on V_8 translates artifact *product-design-specification* from the *initial* or *modified* state to the *completed* state. After the process *verification-and-validation* on V_9 is completed, the process *integration-and-test-plan* on V_{10} and the process *customer-evaluation* on V_{16} (an XX vertex) are enacted.

At the end of a development cycle, the process *customer-evaluation* on V_{16} either chooses (1) an iterate cycle, described as A_{19} (*software* = *evaluated*) or (2) terminates the project, described as A_{20} (*software* = *completed*).

Fig. 10 in the Appendix shows *software's* state transition diagram derived from the example in Fig. 2. The state transition diagram provides managers with a product view so that they can monitor and control projects. A product view gives the current state of a product, which reveals the completeness of the product. SPREM can present various views in multiple layers, where a higher-layered view can hide the details of the lower ones. For example, *software* can be considered as the super-artifact of the artifacts *requirement-specification, product-design-specification, detailed-design-specification, and module*. Each sub-artifact has its own state transition diagram. In this case, if *software* transits from the *prototype-created* state to the *integration-tested* state, this means that the artifacts *detailed-design-specification and model* have been completed.

3.2 A Short Comparison

Petri nets are successful in a wide variety of applications. They have been extended with different notations for different applications, especially timing and functional modeling. Timing is mainly concerned with performance evaluation, including evaluation of the Time Petri Net (TPN) [24], Timed Petri Net, (T_dPN) [28], Stochastic Petri Net (SPN) [22] etc. The functional capability is used to describe various functional specifications of a system. This category of high-level Petri nets includes the Colored Petri Net (CPN) [17], Predicate/Transition net (Pr/T net) [15], Entity/Relation net (ER net) [16] etc. These nets have some similarities and allow tokens to carry various kinds of information.

Some comparisons between SPREM and Petri nets are summarized in Table 1 based on an example Petri net for modeling of the spiral paradigm, shown in Fig. 3.

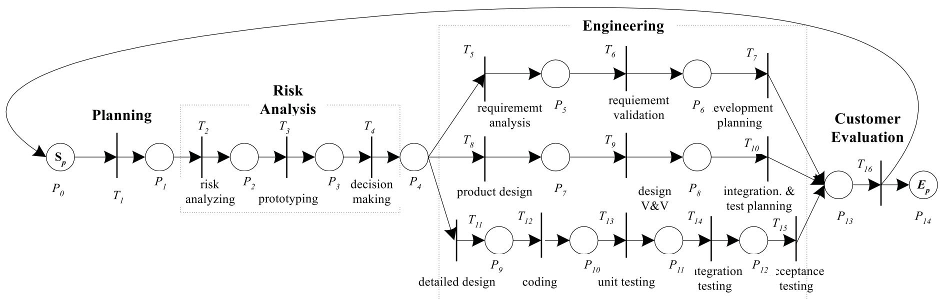

Fig. 3. An example of modeling the spiral model using a Petri Net.

Table 1. Summary of comparisons between SPREM and Petri Nets.

	Original PN	High-level PNs	CPM/PERT	SPREM
Vertex Types	Transition and Place. Any two vertices of the same type can not connect with each other directly.	Transition and Place	AA	AA, AX, XA, and XX
Arc with attributes	No	Yes	AOV or AOA	Artifact state
Token	Yes, without attributes	Yes, with attributes CPN [17], Pr/T net [15], ER net [16], etc.	No	No
Keep History	Yes	Yes	N/A	No, but artifact states maintain the history
Timing	No	TPN [24], T_dPN [28], Stochastic PN [22], etc.	Beta-distribution	Not discussed in this paper.
Precondition Logic	AND for transition OR for place	Complicated Predicate	AND	AND/XOR
Postcondition Logic	AND for transition XOR for place	Complicated Predicate	AND	AND/XOR

Petri nets have two vertex types, transition and place, while SPREM has four: AA, AX, XA, and XX. In a Petri net, a token is added to a place cumulatively when the process on an input transition of the place finishes. The value of an arc in SPREM is set or reset when a process enaction or completion occurs in its head or tail, respectively, no matter what its current value is. SPREM is suitable for modeling a system which focuses on the current status only, but a Petri net is better for a system that needs to keep history information.

SPREM is more concise than Petri nets. In the example of modeling the spiral model, 16 transitions, 14 places, and 33 arcs are used in the Petri net shown in Fig. 3, but 18 vertices and 20 arcs are used in the SPREM graph shown in Fig. 2.

In SPREM, the logic operators in a precondition/postcondition of a vertex used to enact a process are either AND or XOR. In a Petri net, the logic operator is AND-AND for a transition and OR-XOR for a place. Firing a transition T consumes one token from each of T's input places and generates one token which is sent to all of T's output places. For a place P, any of P's input transitions can generate a token sent to P while a token in P can only be consumed by one of P's output transitions. The precondition/postcondition for firing a transition can be a complicated predicate in a high-level Petri net. With SPREM, software project managers can get a more natural view from a graph since the pre/postcondition of enacting a process for a vertex can be seen directly from the SPREM graph.

Furthermore, it is worth noting that SPREM is not intended to serve independently as a software Process Modeling Language PML. Many researches [5, 6, 11, 13] in software process modeling have adopted high-level Petri nets to serve as PMLs. Some comparisons between these PMLs can be found in [1, 14, 23, 31]. To model a software development process, a PML is required to describe various and complicated elements, such as people, resources, artifacts, inter-structures of individual elements, and interrelations between elements, rather than the behavior only [1-3, 8, 9, 14]. SPREM uses simple notation for modeling various behaviors required in software engineering projects such that some management parameters for the projects, such as the enaction capability and schedule, can be evaluated beforehand or during project execution. Also, SPREM is one part of PLAN, a PML used in our related work [18].

4. SOME CHARACTERISTICS OF A SPREM GRAPH

4.1 Enaction Capability

Vertices in a SPREM graph can be divided into four *enactability kinds* according to their enaction capability. A vertex V is called *dead* if V cannot enact a process. Vertex V is called *simple* if V can enact one and only one process. Vertex V is called *essential* if V can enact at least one process. Vertex V is called *dangerous* if it is not one of the above; i.e., it is indeterminate.

A vertex and its successors (predecessors) are not necessarily of the same enactability kind. The enactability kind of a vertex can be determined by several factors: its type (A∗ or X∗), its predecessor types, the values of its input arcs, and its execution behavior (process duration or decision-making).

Obviously, an A∗ vertex is dead if it has one FALSE input arc with a dead tail. An X∗ vertex is dead if (1) it has two or more TRUE input arcs, either of whose tails are dead or ∗A; i.e., these arcs cannot be set to FALSE any more by their tails, or (2) all of its input arcs are FALSE with dead tails.

Whether or not a dangerous vertex can enact a process cannot be determined before project execution. For example, an A∗ vertex V is dangerous if it has one not-dead predecessor whose type is ∗X and may always set its output arc to V to FALSE; i.e., V can not enact any processes. An X∗ vertex is dangerous if it has two or more TRUE input arcs among which there exists at most one whose tail is dead or ∗A.

Let a vertex V have m input arcs A_j, $j = 1, 2, ..., m$, and let each A_j have tail V_j. Suppose vertex V_j enacts processes no more than $MaxEN(V_j)$ times and no less than $MinEN(V_j)$ times.

Consider the case in which V is A∗. V enacts at least one process if each of V's input arcs A_j satisfies one of the following cases: (1) A_j is TRUE and has a ∗A tail, (2) A_j is TRUE and has a dead∗ ∗X tail, or (3) A_j is FALSE and has a simple or essential ∗A tail. The maximum number of processes enacted by V is

$$MaxEN(V) = \min_{1 \le j \le m} MaxEN(V_j) + \delta_j, \text{ where } \delta_j \text{ is equal to one when } A_j \text{ is } TRUE, \text{ zero otherwise.} \quad (1)$$

The minimum number of processes enacted by V is

$$MinEN(V) = \min\left\{\min_{1 \le j \le m} MinEN(V_j) + \delta_j,\ 1\right\}, \text{ if all of V's processors are ∗A.} \quad (2)$$

Consider the case in which V is X∗. The maximum number of processes enacted by V is

$$MaxEN(V) = \begin{cases} \sum_{1 \le j \le m} MaxEN(V_j), \text{ if at most one of } V's \text{ in put arcs } ar\ TRUE; \text{ or} \\ \max\left\{\sum_{1 \le j \le m} MaxEN(V_j) - k + 2,\ 0\right\}, \text{ if } V\ has\ k\ TRUE \text{ in put arcs}; k \ge 2. \end{cases} \quad (3)$$

Consider an X∗ vertex V with all FALSE input arcs. If V has one simple or essential ∗A predecessor V_j, then V's enactability kind is the same as that of V_j. If V has two or more simple or essential ∗A predecessors, then V is essential.

4.2 Enaction Ordering

A vertex in a SPREM graph is said to *depend on* another vertex *graphically* if there exists a path from the latter to the former. A vertex V_1 is absolutely dependent on another vertex V_2 if every path from $\boldsymbol{S_p}$ to V_2 always contains V_1. A vertex V_2 *absolutely trails* another vertex V_1 if every path from V_1 to $\boldsymbol{E_p}$ always contains V_2.

In a SPREM graph, two vertices have no graphical dependency relationship if there is no path between them. Processes enacted in two distinct vertices may run concurrently, and the order of process enaction in separate vertices is not decided only by their graphical dependency. Thus, it is overly complex and unnecessary to compute the order of enaction in separate vertices. To simplify discussion, here, we consider only the first process enaction in a vertex.

Let the *first-enaction* of a vertex be the first time the vertex is enacted. A vertex is *enaction-prior* (*enaction-posterior*) to another if its first-enaction is earlier (later) than that of the other. A vertex is enaction-prior to its A∗ successor; i.e., an A∗ vertex is enaction-posterior to its predecessor. A vertex is enaction-prior to its X∗ successor if there is only one successor. If a vertex is absolutely dependent on another vertex, the latter is always enaction-prior to the former.

The enaction of $\boldsymbol{S_p}$ in a SPREM graph results from an external enaction at the beginning. During the enaction, $\boldsymbol{S_p}$ does not allow any other external enaction until no vertices in the SPREM graph contain running processes. A vertex containing no input paths from $\boldsymbol{S_p}$ is not graphically dependent on $\boldsymbol{S_p}$. A vertex containing no output paths to $\boldsymbol{E_p}$ is not graphically depended upon $\boldsymbol{E_p}$. Two vertices V_1 and V_2, where V_2 is graphically dependent upon V_1, do not necessarily entail V_2 enacting a process when V_1 does so. Neither does it mean that V_1 must enact or complete its process before V_2 can enact its process. However, a vertex that does not depend on $\boldsymbol{S_p}$ graphically can never enact any processes (i.e., it is a dead vertex). A process of a vertex that $\boldsymbol{E_p}$ does not graphically depend on will not affect the project.

4.3 Looping in a SPREM Graph

A *loop* is a simple path in which the first and last vertices are the same. Two paths are said to be *distinct* if they contain no common vertex. A vertex within a loop is called an *entry vertex* to the loop if there is a path from $\boldsymbol{S_p}$ to the vertex that is distinct from the loop except for the vertex. A vertex within a loop is called an *exit vertex* from the loop if there is a path from the vertex to $\boldsymbol{E_p}$ that is distinct from the loop except for the vertex. An input (output) arc of an entry (exit) vertex of a loop is called an *entry* (*exit*) arc of the loop if the arc is not in the loop. A loop may have multiple entry/exit vertices and arcs.

To simplify discussion, each loop in a SPREM graph is allowed to have only one entry vertex, only one exit vertex, and one arc from the exit vertex to the entry vertex of the loop, called the *loop back arc* (*LBA* for short). LBAs are initially set to FALSE. A loop can never make the entry A∗ vertex of the loop enact any process because one input arc (the LBA) is FALSE. A running loop cannot stop if the exit vertex of the loop is ∗A because the LBA will make the entry X∗ vertex enact a process when the exit vertex process is terminated. Thus, the entry and exit vertices of a loop must be X∗ and ∗X, respectively.

A SPREM graph is *well-structured* if and only if all the loop(s) in the SPREM graph contain one X∗ entry vertex, one ∗X exit vertex, and one FALSE LBA. More than one loop may share a common LBA. In a well-structured SPREM graph, a set of loops sharing a common LBA constitutes a strongly connected subgraph, called an *LSUB*. An LSUB can be considered as an XX vertex. Thus, a well-structured SPREM graph can be considered as a partially ordered network, where the duration of the associated project can be estimated beforehand. Like unrestricted *gotos* in programs, general cases of looping are too complicated to deal with [30]. A well-structured SPREM graph is more modularized because every LSUB has only one entry and one exit point. The SPREM graph shown in Fig. 4 is not well-structured since the loop (V_1, A_3, V_3, A_5, V_4, A_6, V_1) has two entry vertices, V_1 and V_2, and the loop (V_3, A_5, V_4, A_7, V_5, A_8, V_6, A_{10}, V_3) has two exit vertices, V_6 and V_7.

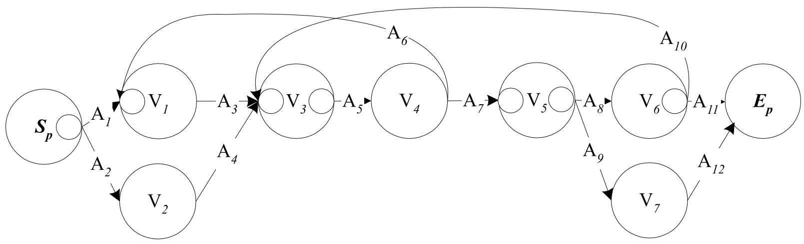

Fig. 4. A SPREM graph that is not well-structured.

4.4. Mutual Exclusion and Dependency

A set of vertices is a Mutually Exclusive Set, MES, if the set has more than one vertex, if only one vertex can enact a process, and if no other enaction is possible. The vertices in an *MES* are called *mutually exclusive* with respect to enaction. In other words, two vertices are mutually exclusive if they are in a common *MES*. A Dependent Enaction Set, *DES*, is a set of vertices in which at least one vertex cannot enact a process.

In an *MES* of two vertices, when one vertex enacts a process, the other cannot. Consider the examples shown in Fig. 5. V_1 and V_2 in (a) are mutually exclusive since V can change one output arc's value once. Cases (b-d) are different. For example, in case (d), V_1 and V_2 enact processes concurrently while A_0 and A_2 are set to TRUE by V_0 and V, respectively.

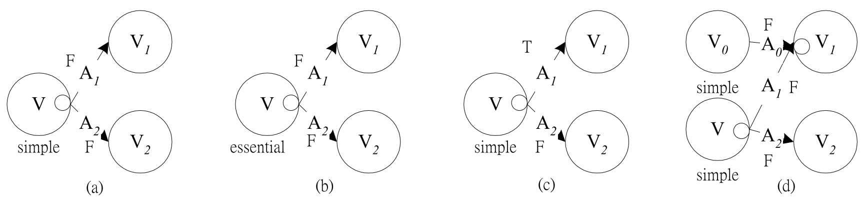

Fig. 5. Examples of mutual exclusion.

Consider the examples shown in Fig. 6. Let all vertices be simple and all arcs be FALSE. Suppose the set of vertices $\{V_1, V_2, V_3\}$ is not a *DES*. Then, the set of vertices $S = \{V_4, V_5, V_6, V_7\}$ in cases (a) and (b) constitutes a *DES*, but it does not in case (c). In (a), at most three vertices in *S* can enact processes. In (b), only V_4, and V_6, V_4, and V_7, or V_5 and V_7 in *S* can enact processes. In (c), all vertices in *S* can enact processes while A_1 is set to TRUE by V_1, A_3 and A_4 are set to TRUE by V_2, and A_6 is set to TRUE by V_3.

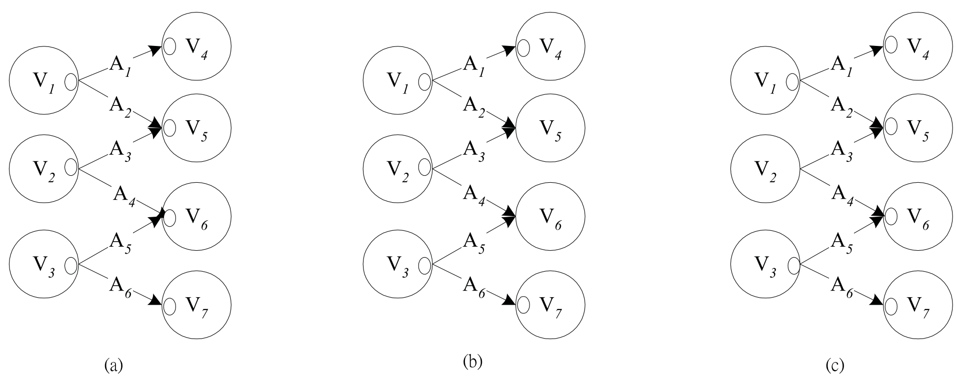

Fig. 6. Examples of a DES.

5. ENACTION CAPABILITY OF ACYCLIC SPREM

5.1 Vertex Enactability

Consider a SPREM graph. Let *LPN*(*V*)/*SPN*(*V*) be the Largest/Smallest Possible Number of processes to be enacted by vertex V. LPN(V) is always larger than or equal to SPN(V). If LPN(V) = 0, V can enact no processes; i.e., V is dead. If LPN(V) = SPN(V) = 1, V can enact one and only one process; i.e., V is simple. If SPN(V) ≥ 1, V can be enact at least one process; i.e., V is essential. If LPN(V) > 0 and SPN(V) = 0, V may or may not enact a process; i.e., V is dangerous. Table 2 summarizes the relationships among the enactability kinds of vertex V and its SPNs and LPNs.

Table 2. Enactability kinds of vertices

	LPN(V) =1	*LPN(V)* =1	*LPN(V)* >1
SPN(V) = 0	Dead	Dangerous	Dangerous
SPN(V) =1	X	Simple	Essential
SPN(V) >1	X	X	Essential

Let an A∗ vertex V have *m* input arcs A_j, $j = 1, 2, ..., m$ and A_j's tail V_j.

V is dead if (1) it has at least one FALSE input arc A_j whose value cannot be changed to TRUE; i.e., A_j is FALSE and V_j is dead; (2) the set of V's predecessors whose output arcs to V are FALSE is a *DES*; or (3) V is mutually exclusive with one of its predecessor(s) that has a FALSE output arc to V.

V is dangerous (i.e., V may or may not enact a process) if V is not dead and has at least one input arc A_j, and if either (1) V_j is ∗X and not dead; or (2) A_j is FALSE and V_j is ∗A and dangerous. In the former case, A_j may be changed to FALSE before V's precondition holds even though it is currently TRUE. In the latter case, A_j may not be changed to TRUE even when V_j is ∗A since V_j may not enact a process.

Otherwise, V can enact at least one process (i.e., V is simple or essential) since each input arc A_j is either (1) TRUE and not changed to FALSE by V_j; or (2) FALSE and will be changed to TRUE. In the former case, A_j is TRUE and V_j is (a) ∗A, or (b) ∗X and dead. In the latter case, A_j is FALSE and V_j is ∗A and simple or essential.

Let an X∗ vertex V have *m* input arcs A_j, $j = 1, 2, ..., m$ and A_j's tail V_j. According to the value of V's input arc, V can be classified into one of the following cases:

Case 1: V has only one TRUE input arc. This case is not allowed in $\boldsymbol{M_0}$.

Case 2: V has two or more TRUE input arcs. If two or more TRUE input arcs are dead or if ∗A tails, then V is dead since these arcs can no longer be set to FALSE by their tails. Otherwise, V is dangerous, and the LPN(V) can be obtained from Eq.(3).

Case 3: All of V's input arcs are FALSE. V is simple or essential if V has at least one simple or essential ∗A predecessor. V is dead if all of V's predecessors are dead. Otherwise, V is dangerous since V has at least one dangerous predecessor or one simple or essential ∗X predecessor. These predecessors will absolutely not change the value of their output arcs to V to TRUE. The LPN(V) of a not-dead vertex can be obtained from Eq.(3). Whether a vertex is simple or essential can be determined as follows:

Case (a): V has one simple or essential ∗A predecessor V_j. V's enactability kind is the same as that of V_j; i.e., SPN(V) = SPN(V_j) and LPN(V) = LPN(V_j).

Case (b): V has two simple or essential ∗A predecessors, namely V_i and V_j. V is essential. If SPN(V_i) = 1 and SPN(V_j) = 1, then SPN(V) = 2, SPN(V) = 1 otherwise.

Case (c): V has more than two simple or essential ∗A predecessors. V is essential and SPN(V) = 1.

5.2 Evaluating an A* Vertex

This subsection discusses how to evaluate the enactability kind of an A∗ vertex. Given a SPREM graph SP and an A∗ vertex V in SP, algorithm *Evaluate-A-Vertex*, presented in Appendix B, can evaluate V's enactability kind as follows. Let each vertex V in a SPREM graph SP be associated with two integer variables *lpn* and *spn* used to store LPN(V) and SPN(V), respectively, and two string variables *kind* and *type* to indicate V's enactability kind and type, respectively. Let each arc A be associated with a Boolean variable *value* used to store A's value and an integer variable *isTrue* used for computing the *lpn* and *spn* of A's head. *A.isTrue* is set to one if A is TRUE, zero otherwise. Before evaluation, in all the vertices and arcs, *V.type*, *A.value*, and *A. isTrue* are defined, and *V.kind* is set to dead and *V. spn* and *V.lpn* are set to zero initially. The evaluation for the A∗ vertex can be done by following the five steps below.

Step 1: Initialization. In step 1, integer variables *MaxEN* and *MinEN* store the results obtained from Eqs.(1) and (2), respectively. The boolean variable *VIsDangerous*, set to FALSE here, indicates whether V has a predecessor that makes it dangerous. The boolean variable *ArcAllTrue*, set to TRUE here, indicates whether all input arcs of V are TRUE.

Step 2: *Evaluate-A-Vertex* uses the function *Has-Exclv-Vex*() [19] to examine whether V has two predecessors that are mutually exclusive or whether V and one of its predecessor(s) are mutually exclusive. If the result is true, i.e., V is dead, it is returned and the algorithm terminates.

Step 3: Each of V's predecessors V_j, j = 1, 2, ..., m, is examined as follows. If one predecessor renders V dead, then it is returned and the algorithm terminates. If one predecessor renders V dangerous, then *VIsDangerous* is set to TRUE. If V is found to have has one FALSE input arc, then *ArcAllTrue* is set to FALSE.

Step 4: *ArcAllTrue* is checked. If *ArcAllTrue* = TRUE, then an error is returned since this is not allowed in $\boldsymbol{M_0}$, and the algorithm terminates.

Step 5: *VIsDangerous* is checked. If *VIsDangerous* = TRUE, then *V.kind* is set to dangerous; otherwise, *V.kind* is set to simple or essential. In step 5.1, *V.lpn* = *MaxENI* is set. *V.spn* = *MinEN* and *V.lpn* = *MaxEN* are set, and V is determined to be simple or essential. If *MaxEN* =1, i.e., V is simple, then *V.kind* is set to simple, or to essential otherwise.

Based on the above, algorithm *Evaluate-A-Vertex* in Appendix B evaluates a vertex V's enactability kind. It first computes LPN(V)/SPN(V) according to Eqs. (1-2), respectively, when the enactability kind of V's predecessors are evaluated. Then, it evaluates whether V is dead in step 2 (mutual exclusive set checking) and in step 3 (checking whether one predecessor renders V dead), and whether V violates the criteria in $\boldsymbol{M_0}$, in step 4. If these cases do not happen, it determines V's enactability kind, as discussed in section 5.1, in step 5. Thus, evaluation is correct.

5.3 Evaluating an X∗ Vertex

This subsection discusses how to evaluate the enactability kind of an X∗ vertex. Given a SPREM graph SP and an X∗ vertex V in SP, the algorithm *Evaluate-X-Vertex* in Appendix B can evaluate V's enactability kind as follows.

Step 1: Initialization. Variables $L1$ and $L2$ store the results obtained from Eq.(3). The variables *NumOfTrueArc* and *NumOfPositiveVex*, set to zero here, represent the number of V's TRUE input arcs and simple or essential ∗A predecessors, respectively. The variable *NumOfDead*, set to zero here, stands for the number of TRUE input arcs in whose tails are either ∗A or dead. The variable *ExistNeuDanVex*, set to FALSE here, indicates whether V has a FALSE input arc whose tail is dangerous or is simple or essential ∗X.

Step 2: Each of V's input arcs A_j and predecessors V_j, $j = 1, 2, ..., m$ is examined. If A_j.value = TRUE, then (1) *NumOfTrueArc* is increased by one, and (2) the algorithm checks whether $V_j.type$ = ∗X or $A_j.kind$ = dead. If the result is TRUE, then *NumOfDead* is increased by one. If $A_j.value$ = FALSE and V_j is simple or essential ∗A, then *NumOfPositiveVex* is increased by one. If V is found to have one dangerous or one simple or essential ∗A predecessor, then *ExistNeuDanVex* is set to TRUE.

Step 3: The algorithm evaluates V in the following cases. In case 1: *NumOfTrueArc* = 1, it returns an error since this case is not allowed in $\boldsymbol{M_0}$. In case 2: *NumOfTrueArc* ≥ 2, it checks *NumOfDead*. If *NumOfDead* ≥ 2, then it returns since V is dead. Otherwise, it sets *V.lpn* = $L1$ and *V.kind* = dangerous. In case 3: *NumOfTrueArc* = 0. If *NumOfPositveVex* = 0 and *ExistNeuDanVex* = TRUE (subcase 1), then *V.kind* = dangerous and *V.lpn* = $L2$ are set (if *ExistNeuDanVex* = FALSE, then V is dead). If *NumOfPositveVex* = 1 (subcase 2), then *V.spn* = V_j*.spn* and *V.kind* = V_j*.kind* are set. If *NumOfPositveVex* = 2 (subcase 3), then *V.kind* = essential is set. If *NumOfPositveVex* ≥ 3 (otherwise part in subcase), then *V.kind* = essential and *V.spn* = 1 are set.

Step 4: The algorithm terminates.

The evaluation correctness for the algorithm *Evaluate-X-Vertex* is discussed in the following. For a vertex V, let the enactability kind of V's predecessors be evaluated. The algorithm first computes variables $L1$ and $L2$ according to Eq.(3). Then, it computes variables *NumOfTrueArc*, *NumOfDead*, *NumOfPositiveVex*, and *ExistNeuDanVex* by evaluating each of V's predecessors in step 2. Based on the above, it correctly determines V's enactability kind according to these variables.

5.4 Evaluating an Acyclic SPREM

Referring to the discussion in section 5.1, V's enactability kind can be determined if those of its predecessors are known. A topological sorting algorithm lists the vertices in an acyclic graph with the predecessors at the front. In an acyclic SPREM graph, the algorithm *Evaluate-Acyclic-SPREM* in Appendix B can be used to traverse an acyclic SPREM graph *SP* to output the LPN, SPN, and enactability kind of each vertex.

Let each vertex V in *SP* be associated with two integer variables *lpn* and spn storing LPN(V) and SPN(V), respectively, and two string variables *kind* and *type* indicating V's kind and type, respectively. Let each arc A in *SP* be associated with a Boolean attribute *beenTraversed*. The *beenTraversed* of an arc, set to FALSE initially, is changed to TRUE when its tail is traversed. Let each arc A be associated with a Boolean variable *value* used to store A's value, and an integer variable *isTrue* used for computing the *lpn* and *spn* of A's head. A.*isTrue* is set to one if A is TRUE, zero otherwise.

Evaluate-Acyclic-SPREM uses a queue *Q* to store vertices temporarily during traversal of *SP*. A vertex V is inserted into *Q* when all of its input arcs have *beenTraversed* = TRUE; i.e., the computation of all its predecessor(s) is finished. V is removed from *Q* while it is being computed. Each vertex can be computed at most once as follows.

Step 1: Initialization. Queue *Q* is cleared. For each vertex V in *SP*, the variable *V.kind* is set to dead, and *V.spn* and *V.lpn* are set to zero. For each arc A, *A.beenTraversed* is set to FALSE, and *A.isTrue* is set to one if *A.value* is TRUE, zero otherwise.

Step 2: Traversing *SP* from $\boldsymbol{S_p}$. First, $\boldsymbol{S_p}$*.kind* = simple, LPN($\boldsymbol{S_p}$) = 1, and SPN($\boldsymbol{S_p}$) = 1 are set, and *beenTraversed* in all of the output arcs of $\boldsymbol{S_p}$ is set to TRUE. Then, $\boldsymbol{S_p}$'s successor(s) are inserted into *Q* if all of their input arcs where *beenTraversed* = TRUE.

Step 3: Performing traversal with a ***while*** loop. The loop is repeated until *Q* is empty, i.e., until all vertices are traversed. Then the algorithm terminates. In each turn, a vertex V is retrieved from queue *Q* and evaluated. If *V.type* = A∗, then *Evaluate-A-Vertex* is invoked, *Evaluate-X-Vertex* otherwise. Then, *beenTraversed* in all output arcs of V is set to TRUE. If V's successor has input arcs where *beenTraversed* = TRUE, then it is inserted into *Q*.

The algorithm *Evaluate-Acyclic-SPREM* traverses acyclic SPREM graph *SP* in topological sorting order using queue *Q*. This order allows the predecessors of a vertex V to be visited before V; i.e., when V is evaluated, its predecessors have already been evaluated. When V is evaluated, either algorithm *Evaluate-A-Vertex* or algorithm *Evaluate-X-Vertex* is used. Thus, evaluation is correct.

5.5 An Example of Evaluating an Acyclic SPREM

A SPREM graph is *well-defined* if and only if the network contains no dead vertices. Fig. 7 shows a SPREM graph that is not *well-defined*. Vertex V_4 is dead since it has two TRUE input arcs A_4 and A_5 whose tails are simple, e.g., V_1 and V_2, respectively. V_8 is dangerous since it has two TRUE input arcs, A_{12} and A_{13}, in which A_{12}'s tail V_5 is ∗X. Table 3 lists the $\boldsymbol{M_0}$ of the SPREM graph. Table 4 shows the results of analyzing the SPREM graph using *Evaluate-Acyclic-SPREM*.

Table 3. The M_0 of the SPREM graph in Fig. 7.

Arcs	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9
M_0	F	F	F	T	T	F	F	F	F
Arcs	A_{10}	A_{11}	A_{12}	A_{13}	A_{14}	A_{15}	A_{16}	A_{17}	A_{18}
M_0	T	F	T	T	F	T	F	T	F

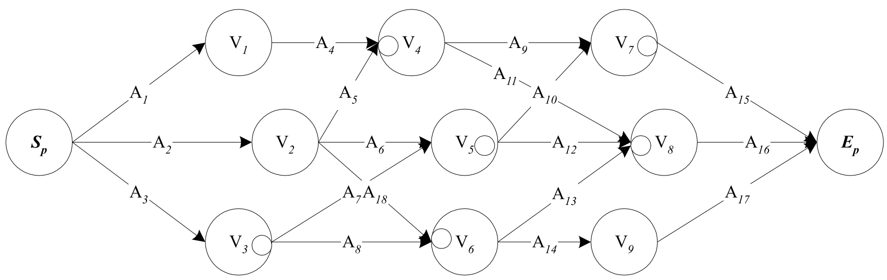

Fig. 7. A SPREM graph that is not well-defined.

Table 4. The results of analyzing the SPREM graph in Fig. 7.

Vertex	Kind	SPN	LPN	Vertex	Kind	SPN	LPN
S_p	Simple	1	1	V_6	Essential	1	2
V_1	Simple	1	1	V_7	Dead	0	0
V_2	Simple	1	1	V_8	Dangerous	0	3
V_3	Simple	1	1	V_9	Essential	1	2
V_4	Dead	0	0	E_p	Dangerous	0	1
V_5	Dangerous	0	1				

6. ENACTION CAPABILITY OF CYCLIC SPREM

In a SPREM graph, general loopings, like arbitrary *goto* in a program, are too complicated to deal with. Unrestricted loopings should be avoided since they make a program difficult to understand and maintain. In a well-structured SPREM graph, each loop has only one entry and one exit. A set of loops sharing a common entry and exit can be deemed as a group of iterations. A loop can be nested within another one. Therefore, the set of vertices within a loop can be reduced as one higher-level vertex to help: (1) by recognizing a well-structured SPREM graph as a partially ordered network, where the duration of the associated project can be estimated at the beginning[3], (2) by reducing the analysis complex-

[3] The estimation of a project's duration is beyond the scope from this paper. For more details, please refer to [9].

ity of a complicated system represented by SRPEM, and (3) making a SPREM graph easy to understand and maintain. Algorithm *Is-Structured* [19] can be used to determine whether a SPREM graph is well-structured.

An LSUB can be reduced to an XX vertex whose input and output arcs are those of the entry and exit of the LSUB, respectively, except for the LBA. The *inner* of an LSUB consists of all the vertices inside the LSUB except for the entry and exit. An LSUB is called a *containing LSUB* of a vertex V if V is the entry, exit, or an inner of the LSUB. Fig. 8 shows a sample LSUB in (a) and its corresponding XX vertex $V_{a\text{-}b}$, in (b). The LSUB in is denoted as LSUB(V_a, V_b). A nested LSUB can eventually be reduced recursively to an XX vertex eventually. Fig. 9 shows a sample nested LSUB. LSUB(V_b, V_c) in (a) is reduced to $V_{b\text{-}c}$ in (b), and LSUB(V_a, V_d) in (b) is reduced to $V_{a\text{-}d}$ in (c).

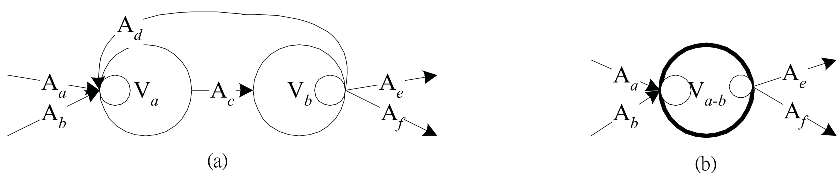

Fig. 8. A sample LSUB.

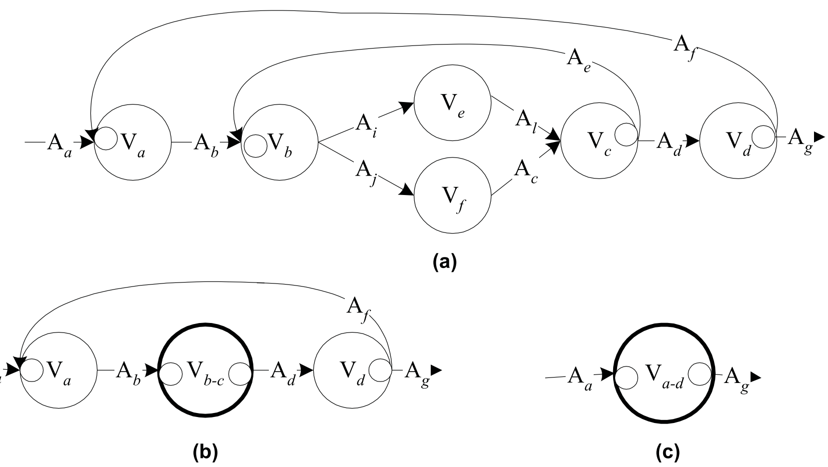

Fig. 9. A sample nested LSUB.

The enaction capability of a well-structured SPREM *SP* graph can be evaluated by: (1) removing LBAs on each LSUB in *SP* to make the SP acyclic, (2) evaluating the acyclic *SP* with *Evaluate-Acyclic-SPREM*, and (3) adding the LBAs back to *SP* to refine the evaluation. In a well-structured *SP*, the entry and exit vertices of an LSUB are X∗ and ∗X, respectively, and the LBAs are set to FALSE initially. Thus, the enactability kind of a vertex in LSUB is not changed by the refinement, as mentioned in the earlier discussion about *Evaluate-X-Vertex*.

7. CONCLUSIONS AND FUTURE WORK

This paper has presented the software project review and evaluation model SPREM. SPREM can be used to express processes with sequence, parallelism, iteration, synchronization, and decision-making behaviors in software projects. Its modeling capability has been demonstrated using an example which describes the *spiral model* in software engineering. SPREM is a superset of CPM/PERT and has more concise and natural graphical notation than do (high-level) Petri nets. Several behavioral properties of SPREM have been discussed. For example, its enaction capability can be used to evaluate the possibility that a vertex will enact processes beforehand. Project managers can revise a SPREM graph in order to rescue dead vertices before project execution. Furthermore, enaction ordering allows project managers to calculate the dependency between the processes to be enacted. This might help in computing important information, such as the critical paths among these processes. The detection of MES and DES in a SPREM graph can help project managers avoid pitfalls in modeling. The algorithms *Evaluate-Acyclic-SPREM* and *Evaluate-Cyclic-SPREM*, based on several other algorithms, including *Evaluate-A-Vertex*, *Evaluate-X-Vertex* etc., have been presented to evaluate the enaction capability of acyclic and well-structured SPREMs, respectively.

In addition, some comparisons between SPREM and Petri nets have been discussed in this paper. The high-level Petri nets are better for modeling a system that needs to keep history information, and they can express a complicated predicate in order to enact a transition. However, SPREM is concise and better for modeling a system where the focus is on the current status only. With SPREM, software project managers can get a more natural view from a graph. Some further works need be done in the future:

(1) analysis of the enaction capabilities for a general SPREM graph,
(2) detection of the possible execution scenarios of a SPREM graph, and
(3) estimation of the completion time (or cost) for a SPREM graph, including determination of the critical path(s) and the instantiating, running, and terminating times for each process.

REFERENCES

1. P. Armenise, S. Bandinelli, C. Ghezzi, and A. Morzenti, "A survey and assessment of software process representation formalisms," *International Journal of Software Engineering and Knowledge Engineering,* Vol. 3, No. 3, 1993, pp. 401-426.
2. J. W. Armitage and M. I. Kellner, "A conceptual schema for process definitions and models," in *Proceedings of 4th International Conference on Software Processes*, 1994, pp.153-175.
3. R. Balzar, "What we do and don't know about software process," *CQS European Observatory on Software Engineering: CASE and Software Quality*, October 1990.
4. B. W. Boehm, "A spiral model of software development and enhancement," *ACM SIGSOFT Software Engineering Notes*, Vol. 11, No. 4, 1986, pp.14-24.
5. S. Bandinelli, A. Fuggetta, and C. Ghezzi, "Software process as real time systems: a case study using high-level Petri nets," in *Proceedings of International Phoenix Conference on Computer and Communication*, 1994, pp. 231-242.

6. S. C. Bandinelli and A. Fuggetta, "Software process model evolution in the SPADE environment," *IEEE Transaction on Software Engineering*, Vol. 19, No. 12, 1993, pp. 1128-1144.
7. S. Bandinelli, A. Fuggetta, L. Lavazza, M. Loi, and G. P. Picco, "Modeling and improving an industrial software process," *IEEE Transaction on Software Engineering*, Vol. 21, No. 5, 1995, pp. 440-454.
8. B. Curtis, M. I. Kellner, and J. Over, "Process Modeling," *Communication of ACM*, Vol. 35, No. 9, 1992, pp. 75-90.
9. C. W. Dawson and R. J. Dawson, "Clarification of node representation in generalized activity networks for practical project management," *International Journal of Project Management*, Vol. 12, No.2 1994, pp. 81-88.
10. S. E. Elmaghraby, "An algebra for the analysis generalized activity networks," *Management Science*, Vol. 10, No. 3, 1964, pp. 494-514.
11. W. Emmerich and V. Gruhn, "FUNSOFT Nets: A Petri-Net based software process modeling language," in *Proceedings of the 6th International Workshop on Software Specification and Design*, 1991, pp. 175-184.
12. H. Esiner, "A generalized network approach to the planning and scheduling of a research project," *Operation Research*, Vol. 10, No. 1, 1962, pp. 115-125.
13. C. Fernstrom, "Process weaver: adding process support to UNIX," in P*roceedings of 3rd International Conference on Software Process*, 1993, pp. 2-11.
14. A. Fuggetta and C. Ghezzi, "State of the art and open issues in process-centered software engineering environment," *Journal of System Software*, Vol. 26, No. 1, 1994, pp. 53-60.
15. H. J. Genrich, "Predicate/transition nets," in *Advances in Petri Nets*, 1986, W. Brauer, W. Reisig, and G. Rozenberg, Eds. New York: Springer-Verlag, 1987, pp. 3-43.
16. C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezze, "A unified high-level Petri net formalism for time-critical systems," *IEEE Transaction on Software Engineering*, Vol. 17, No. 2, 1991, pp. 160-172.
17. K. Jensen, "Colored Petri nets and the invariant method," *Theoretical Computing Science*, Vol. 14, No. 3, 1981, pp. 317-336.
18. B. S. Liang, M. F. Chen, and F. J. Wang, "A distributed software process engineering environment," in *Proceedings of Workshop of Distributed System Technology and Application*, 1997, Taiwan, R.O.C., pp. 531-538.
19. B. S. Liang and F. J. Wang, "A software project review and evaluation model, SPREM," Technical Report CSIE-97-1003, Department of Computer Science and Information Engineering, National Chiao-Tung University, March 1997.
20. L. C. Liu and E. Horowitz, "A formal model for software project management," *IEEE Transaction on Software Engineering*, Vol. 15, No. 10, 1989, pp. 1280-1293.
21. J. Magott and K. Skudiarski, "Combing generalized stochastic Petri networks for the performance evaluation of concurrent process," in *Proceedings of IEEE TH*0288-1/89/0000/0249, 1989, pp. 249-256.
22. M. A. Marsan, G. Balbo, and G. Gonte, "A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor system," *ACM Transaction on Computer System*, Vol. 2, No. 2, 1984, pp. 93-122.
23. I. R. McChesney, "Toward a classification scheme for software process modeling approach," *Information and Software Technology*, Vol. 37, No. 7, 1996, pp. 363-374.
24. P. Merlin and D. J. Faber, "Recoverability of communication protocol," *IEEE Transac-*

tions on Communication, Vol. 24, No. 9, 1976, pp. 1036-1045.
25. C. S. Park, G.S. Lee, and J. M. Yoon, "An executable software project management model by beta-distributed stochastic Petri nets," in *Proceedings of IEEE TENCON (Region 10 Conference)*, 1993, pp. 430-434.
26. R. S. Pressman, *Software Engineering: A Practitioner's Approach*, McGRAW-Hill International, 1993, pp. 29.
27. A. A. B. Pritsker and G. E. Whitehouse, "GERT: Graphical evaluation and review technique part I: fundamentals," *Journal Industry Engineering*, Vol. 17, No. 3, 1966, pp. 267-274.
28. C. Ramchandani, "A study of asynchronous concurrent systems by timed Petri nets," Technical Report, 120, Project MAC, Massachusetts Institute Technology, February 1974.
29. C. V. Ramamoorthy and G. S. Ho, "Performance evaluation of asynchronous concurrent systems by Petri nets," *IEEE Transactions on Software Engineering*, Vol. 6, No. 5, 1980, pp. 440-449.
30. R. W. Sebesta, *Concepts of Programming Languages*, Addison-Wesley, 1996, pp. 309-313.
31. I. Thomas, "The strengths and weaknesses of process modeling formalisms," in *Proceedings of 7th International Software Process Workshop*, 1991, pp. 2-9.

APPENDEX−A

Table 5. The vertex processes in the example shown in Fig. 2.

Vertex	Process
V_1	determine objectives, alternatives, constraints
V_2	risk analysis
V_3	prototyping
V_4	decision making based on simulations, model, benchmarks
V_5	requirement analysis
V_6	requirement validation
V_7	development planning
V_8	product design
V_9	design verification and validation
V_{10}	integration and test planning
V_{11}	detailed design
V_{12}	coding
V_{13}	unit testing
V_{14}	integration testing
V_{15}	acceptance testing
V_{16}	customer evaluation

Table 6. The vertex arcs in the example shown in Fig. 2.

Arc	Artifact = State
A_1	software = initialized
A_2	software = planned
A_3	software = risk-analyzed
A_4	software = prototype-created
A_5	requirement-specification = initialized or modified
A_6	requirement-specification = completed
A_7	requirement-specification = validated
A_8	software = development-planned
A_9	product-design-specification = initialized or modified
A_{10}	product-design-specification = completed
A_{11}	product-design-specification = validated
A_{12}	software = integration & test-planed
A_{13}	detailed-design-specification = initialized or modified
A_{14}	detailed-design-specification = completed
A_{15}	module = coded
A_{16}	module = tested
A_{17}	software = integration-tested
A_{18}	software = acceptance-tested
A_{19}	software = evaluated
A_{20}	software = completed

APPENDIX – B : ALGORITHMS

B -1

Algorithm *Evaluate-A-Vertex* (*SP, V*)
Input: $SP = < \boldsymbol{V}, \boldsymbol{A}, \boldsymbol{S_p}, \boldsymbol{E_p}, \boldsymbol{M_0} >$ (a SPREM grapg), $V \in \mathbf{V}$, is an A* vertex.
Output: *V.spn, V.lpn*, and *V.kind*.

/ Given a SPREM graph SP and an A* vertex V in SP, output V's SPN, LPN, and enactability kind. Let V have m input arcs A_j, j = 1 to m, and let each A_j have a tail V_j.*
Suppose that V.kind is set to dead, that V.spn and V.lpn are set to zero initially, and that V's predecessors have been evaluated already./*

begin

step 1: *MaxEN* = $\min_{1 \le j \le m} V_j.lpn + V_j.is\ True$;/* *equation (1)*/

$MinEN = \min\left\{\min_{1 \le j \le m} V_j.spn + V_j.isTrue,\ 1\right\}$; /* *equation (2)* */

VIs Dangerous = FALSE; *ArcAllTrue* = FALSE;

step 2: if *Has-Exclv-Vex (V)*
then return; /* *V.kind = dead; V.SPN = 0; V.LPN = 0*/*
step 3: for *j* = 1 **to** *m* **do**
begin
step 3.1: if ($A_j.value$ = FALSE and $V_j.kind$ = dead)
then return; /* *V.kind = dead; V.SPN = 0; V.LPN = 0* */
step 3.2: if not *VIsDangerous* **then**
begin
if ($V_j.type$ = *X and $V_j.kind$ ≠ dead) or
($A_j.value$ = FALSE and $V_j.type$ = *A and $V_j.kind$ = dangerous)
then *VIsDangerous* = TRUE;
end; /* *of if* */
step 3.3: if $A_j.value$ = FALSE then *ArcAllTrue* = FALSE;
end; /* *of for* */
step 4: if *ArcAllTrue*
then return error; /* *this case is not allowed in* $\boldsymbol{M_0}$*/
step 5: if *VIsDangerous*
step 5.1: then begin
V.kind = dangerous; *V.lpn = MaxEn;*
end;
step 5.2: else begin */∀ V_j *satisfy one of the following cases:*
(1) A_j *is TRUE and* V_j *is *A,*
(2) A_j *is TRUE and* V_j *is *X and dead, or*
(3) A_j *is FALSE,* V_j *is *A and simple or essential.* */
V.lpn = MaxEN; V.spn = MinEN;
if *(MaxEN* =1*)*
then *V.kind = simple* **else** *V.kind = essential;*
end if;
end; */ *of Evaluate-A-Vertex* */

B-2

Algorithm *Evaluate-X-Vertex (SP, V)*
Input: SP = <$\boldsymbol{V, A, S_p, E_p, M_0}$> (a SPREM), V ∈ **V**, is an A* vertex.
Output: *V.spn, V.lpn,* and *V.kind.*
/* *Given a SPREM SP and an X* vertex V in SP, output V's SPN, LPN, and kind. Let V have m input arcs* A_j, *j* = 1 *to m, and let each* A_j *have a tail* V_j. *Suppose that V.kind is set to dead, that V.spn and V.lpn are set to zero initially, and that V's predecessor have been evaluated already.*

```
begin
step 1: L1 = max { Σ_{1≤j≤m} V_j.lpn − k + 2, 0 }, where k is the number of V's TRUE input arcs'/*
            equation (3) */
        L2 = Σ_{1≤j≤m} V_j.lpn /* equation (3) */
        NumOfTrueArc = 0;
        NumOfDead = 0;
        NumOfPositveVex = 0;
        ExistNeuDanVex = FALSE;
step 2: for j = 1 to m do
        begin
step 2.1: if (A_j.value = TRUE)
step 2.2: then begin;
                    NumOfTrueArc = NumOfTrueArc +1;
                    if (V_j.type = * A or V_j.kind = dead)
                      then NumOfDead = NumOfDead + 1;
                    end;
step 2.3: else begin  /* A_j = FALSE */
                    if ((V_j.type = * A) and (V_j.kind = simple or V_j.kind = essential))
                      then NumOfPositveVex = NumOfPositveVex +1;
                    if (not ExistNeuDanVex) then
                    begin
                      if (V_j.kind = dangerous) or
                        ((V_j.type = *X) and (V_j.kind = simple or V_j.kind = essential))
                        then ExistNeuDanVex = TRUE;
                    end;
                  end; /* of if */
            end; /* of for */
step 3: case NumOfTrueArc of
step 3.1: case 1: = 1 /*only one A_j = TRUE */
              return error; */ This case is not allowed in M_0*/
step 3.2: case 2: ≥ = 2 /*two or more A_j = TRUE */
              if (NumOfDead < 2) /* NumOfDead ≥ 2 then V is dead */
              then V.kind = dangerous; V.lpn = L1; else V.kind = dead;
step 3.3: otherwise: /* NumOfTrueArc = 0, i.e., ∀A_j = FALSE */
              V.lpn = L2;
              case NumOfPositveVex of
              case 1: = 0; /* no positive predecessor */
                if (ExistNeuDanVex = TRUE) /* else ∀V_j are negative, V is dead */
                  then V.kind = dangerous;
              case 2: = 1; /* only one positive predecessor */
                V.spn = V_j.spn; kind = V_j.kind
              case 3: = 2; /* exactly two positive predecessors */
                /* suppose the two predecessors are V_i and V_j */
                  V.kind = essential;
                if V_i.spn = 1 and V_j.spn =1
```

then *V.spn* = 2 **else** *V.spn* = 1;
otherwise: /* ≥ 3, *i.e., three or more positive predecessors* */
V.kind = essential; *V.spn* = 1;
end case;
end case;
step 4: return;
end; */ *of Evaluate-X-Vertex*/*

B-3

Algorithm *Evaluate-Acyclic-SPREM (SP)*
Input: SP = <***V, A,*** $\boldsymbol{S_p}$, $\boldsymbol{E_p}$, $\boldsymbol{M_0}$ > (a SPREM).
Output: *V.spn, V.lpn,* and *V.kind.* for each V in SP.
/* *Given a SPREM SP, for each V in SP, output V's SPN, LPN, and enactability kind.* */
begin
/* *initialization* */
step 1: clear queue *Q;*
for each vertex *V in* ***V*** **do**
begin
V.kind = dead; */ *suppose V.type is defined in SP* */
V.spn = 0;
V.lpn = 0;
end;
for each arc A *in* ***A*** **do**
begin
A.beenTraversed = FALSE;
if *A.value* = TRUE /* *suppose A.value is defined in SP* */
then *A.isTrue* =1
else *A.isTrue* = 0;
end;
/* *traverse SP from* $\boldsymbol{S_p}$ *firstly* */
step 2: $\boldsymbol{S_p}$.*kind* = simple; $\boldsymbol{S_p}$.*lpn* = 1; $\boldsymbol{S_p}$.*spn* = 1;
set the *beenTraversed* of all output arcs of $\boldsymbol{S_p}$ to TRUE;
for each successor V_m of $\boldsymbol{S_p}$ whose all input arcs with *beenTraversed* = TRUE **do**
En-Queue (Q, V_m*);*
end;
/* *traverse the vertices in topological order* */
step 3: while not *Empty (Q)* **do**
begin
step 3.1: *V = De-Queue(Q)*;
/* *evaluate a vertex* */
step 3.2: if *V.type* = A*
then *Evaluated-A-Vertex (V)*
else *Evaluated-X-Vertex (V);* /* *V.type* = X * */

step 3.3: set the *beenTraversed* of all *output* arcs of V to TRUE;
step 3.4: for successor V_m of V whose all input arcs with *beenTraversed* = TRUE **do**
En-Queuye(Q, V_m);
end;
end; /* *of while* */
end; /* *of Evaluate-SPREM* */

Bin-Shiang Liang (梁賓先) received his B.S. and M.S. degrees in computer science and information engineering from National Chiao Tung University in 1991 and 1993, respectively. He is currently a Ph.D. candidate in computer science and information engineering at National Chiao Tung University. His research interests include software measurement, software process modeling, process-centric software engineering environment (PSEE), network-centric computing (NCC), ISO 900 standards certification, and Web-based information systems (WIS).

Jenn-Nan Chen (陳振楠) received his B.S. degree in Applied Mathematics from the Chung Cheng Institute of Technology, Taiwan, R.O.C., in 1972. He received his M.S. degree from the Department of Operation Research, Stanford University, U.S.A., in 1976. He then worked at the Chung Shan Institute of Science and Technology Computer Center as a system engineer for seven years. He attended Northwestern University in the spring of 1983 and received his M.S. and Ph.D. degrees in Computer Science from the Department of Electrical Engineering and Computer Science in 1984 and 1986. He is currently the president of the SAMAR Techtronics Corporation. His interests include software engineering, software quality assurance, software process modeling, object-oriented techniques, and ISO9000 software methodology development.

Feng-Jian Wang (王豐堅) graduated from National Taiwan University, Taipei, Taiwan, R.O.C., in 1980. He received his M.S. and Ph.D. degrees in E.E.C.S. from Northwestern University, U.S.A., in 1986 and 1988. He is currently a professor in the Department of Computer Science and Information Engineering, National Chiao Tung University. His interests include software engineering, compiler, object-oriented techniques, distributed system software, and the internet.