A Formal Model for Software Project Management

LUNG-CHUN LIU AND ELLIS HOROWITZ

Abstract—**A model called DesignNet for describing and monitoring the software development process is presented. This model utilizes the AND/OR graph and Petri net notation to provide the description of a project work breakdown structure and the specification of relationships among different project information types (activity, product, resource, and status report information). Tokens are objects with specific properties. Token propagation through structural links allows aggregate information to be collected automatically at different levels of detail. The transition firing is a nonvolatile process and creates new token instances with time dependent information. The typed places, together with connections among them, defines the static construct of a project. Whenever transitions are fired, the project execution history is recorded by the token instances created.**

Using the model, we have provided definitions for basic properties of a successful project, namely connectedness, plan complete, plan consistent, and well-executed. We have given algorithms for computing these functions and shown that the computing time is linear in the size of the project. This assures that any system based on DesignNet should be able to compute these functions efficiently. Finally, we have shown how the waterfall life cycle model maps onto a DesignNet and the implications for project planning, cost estimation, project network construction, reinitiation of activities, and traceability across the life cycle. Other life cycle models can be equally treated.

Index Terms—**Software development, software project management.**

I. The Software Process Model

RECENTLY there has been a great deal of discussion and concern about the lack of an appropriate model for the development of large-scale software [1], [4], [7], [13], [19]. Why should we attempt to develop models of the software development process? One reason is that by having a model we can understand for ourselves and explain to others the various steps that we must go through before completing a project. So a model is a means of communication between the developers and the customer and between the developers themselves. Another use of a model is for assisting in the management of the process. A model can provide management with a set of points (milestones), that can be examined to determine the rate of progress of the project. A third advantage of a model is that it gives software engineers a foundation for building tools that will support and enhance the software process.

The traditional model for software development is the so-called waterfall model. A complete picture of this model is found in [19, pp. 338] and as it has been reprinted so often we will not reproduce it here. The waterfall model views software development as a manufacturing process. Each step is a phase, and the completion of one phase leads to another. Each phase has inputs from a previous phase and outputs (some of which are deliverables), that it produces. These outputs are used by management for tracking the progress of a project. For example, Royce lists six types of documents as outputs:

- software requirement document
- preliminary design specification
- interface design specification
- final design specification
- test plan
- operating instruction manual.

The waterfall model is often shown with back pointing arrows as well as forward pointing arrows, acknowledging that the manufacturing model captured in the waterfall chart is not precise, and that previous phases may be returned to. Royce also emphasizes that the waterfall chart is not intended to preclude prototyping.

Criticisms of the waterfall model have appeared in several places, e.g., [4], [13], who say that

- it is foolish to believe that one model is appropriate for *all* software development projects,
- there is inadequate modeling of the fact that requirements change,
- there is no modeling support that involves the end users in the development process,
- it fails to treat software development as a problem solving process and therefore offers little insight into the actions and events that precede the finished products.

Agresti develops three alternative models, called 1) prototyping, 2) operational specifications, and 3) transformational implementation. After studying these alternatives, Curtis *et al.* conclude that these models are based on technologies that are new and unproven and hence it is unclear whether they will scale up to industrial size.

It is our belief that a single model may not exist which will perfectly describe the software development process from all angles. However, for the purpose of this paper we would highlight these half dozen features that we feel are essential.

Manuscript received February 8, 1988; revised July 29, 1988. Recommended by W. Royce.

L.-C. Liu was with the Department of Computer Science, University of Southern California, Los Angeles, CA 90089. He is now with Cadence Design Systems, Inc., Santa Clara, CA 95054.

E. Horowitz is with the Department of Computer Science, University of Southern California, Los Angeles, CA 90089.

IEEE Log Number 8930133.

0098-5589/89/1000-1280$01.00 © 1989 IEEE

1) The model must adequately describe the fact that software development is a *design process*. Design is inherently an evolutionary process where steps are continually returned to, and moreover, sometimes steps are entirely abandoned and new steps inserted.

2) The model should be able to include the fact that a *large-scale* software development project is inherently a parallel process with many people undertaking tasks simultaneously.

3) The model should be able to indicate that a diverse set of conditions must exist before an activity can be undertaken.

4) The model should be able to indicate all artifacts that are produced at various points in the process.

5) If an activity fails, the model should be able to indicate the activities and resources that are affected. Affected activities may have to be re-executed.

6) The model should be able to indicate the extent and nature of resources involved in a subtask, including people, consumable, and nonconsumable resources.

In this paper we propose a model that attempts to satisfy all of these criteria. This model is a hybrid model which utilizes AND/OR structure operators to describe the work breakdown structure and Petri net notation to represent the dependencies among activities, resources, and products. It not only provides the project staff with a structural view of the project information, but it also assists the project manager in monitoring progress. Before the model is formally presented, we will briefly review the basic definitions of project management and their relation to software projects. Then AND/OR graphs and Petri nets are surveyed.

II. The Traditional Elements of Project Management

A. Basic Definitions of Project Management

A *project* is composed of a series of activities directed to the accomplishment of a desired objective which usually results in the delivery of a system or product. A project will generally have a minimum set of features including:

- a specific objective to be completed within certain specifications
- defined start and end dates
- funding limits (budget)
- consumption of resources (i.e., money, people, equipment).

An *activity* is a task with a well-defined beginning and a well-defined end that can be performed by a single functional entity. An *event* (also called *milestone*) is an occurrence at a point in time that signifies the start or completion of one or more tasks or activities.

A *work break-down structure* (*wbs*) is a graphic portrayal of the project, exploding it in a level-by-level fashion down to the degree of detail needed for effective planning and control. It must include all deliverable end items (equipment, facilities, services, manuals, reports, and so on) and the major functional tasks that must be performed. An example *wbs* taken from a software project is shown in Fig. 1. The root node is decomposed into six major tasks according to steps in the traditional software development life cycle.[1] Several tasks are further decomposed into the next level tasks. Typically many tasks will extend several levels deep, but only a few levels are shown here for simplicity.

Project management is the mixture of people, resources, systems, and techniques required to carry the project to successful completion, see, e.g., [12], [2]. The goal of project management is to accomplish the project before the designated deadline, within the budget, and utilizing the existing resources efficiently. Symptoms caused by poor project management include: late completion, penalties, cost overruns, project staff turnover, duplication of effort, and inefficient use of functional specialists. Innovation of project management tools is especially crucial for *design projects* which are characterized as applying new technologies to develop new products in an uncertain environment.

To make these definitions concrete, consider Fig. 1 again. The sample project is: *to develop a spreadsheet package*. A sample task is: *to design and build a plotter driver for the spreadsheet package*. This task is broken into a sequence of tasks that extends across the *wbs* in the following order:

1) The format of the graphics command file is determined as an activity of the *system specification* task.

2) Once step 1 is completed, an experienced system programmer, a plotter programming manual, and funding, are available, the *plotter driver design task* can be initiated.

3) After design is completed, the *coding phase* can begin.

4) When the driver is finished and test material has been prepared, *functional testing* can be started.

5) At any time, if the graphic command format (specification) needs to be changed, return to step 2.

6) If functional testing shows any failure, return to step 3 and redesign the plotter driver.

7) If the plotter malfunctions, submit a proper request to the equipment support department and temporarily suspend the testing activity. The testing staff may be assigned to other tasks during this period.

There are several deficiencies one can identify in attempting to use the *wbs* for modeling a software project.

- There is no provision for expressing resources (such as a plotter manual and programmer) that must be available.
- There is no explicit representation of the fact that many activities are being pursued concurrently.
- There is no provision for indicating when an activity is re-executed, as described in steps 5 and 6, or suspended, as in step 7.

[1] A *wbs* for recently proposed alternate life cycle definitions is easily constructed.

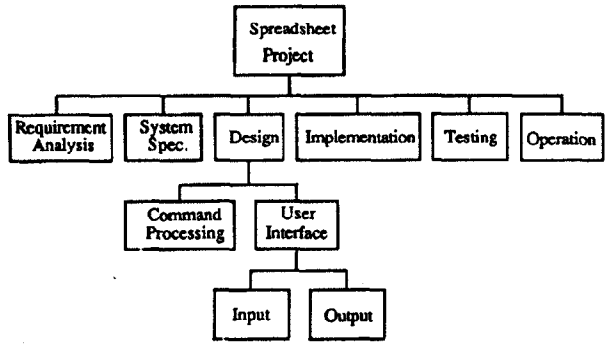

Fig. 1. A sample work breakdown structure.

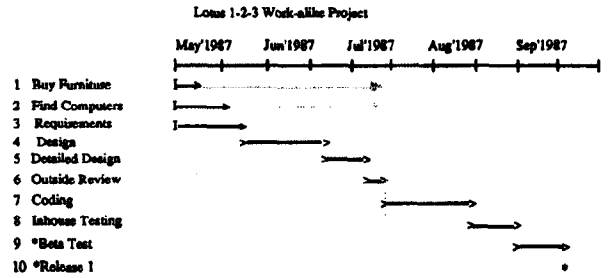

Fig. 2. A sample Gantt chart.

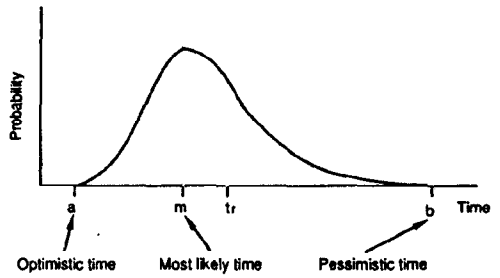

Fig. 3. Beta distribution for PERT activities.

- There is no provision for removing activities and adding new ones.

B. The Traditional PERT/Gantt Models

There are several traditional project planning and control models that are used and it is instructive to examine them [8], [6].

The Gantt Model: In this scheme project activities are represented as lines on a calendar oriented chart, with special marks to indicate major milestones and activities that cannot be delayed without delaying the entire project (critical activities). In Fig. 2 you see a sample of a Gantt chart. Some special features to notice are: the time scale on the top line, how the critical activities are highlighted with double bars, and how slack time is shown as a series of dots. See [6] for more details.

The Critical Path Method (CPM): This approach emphasizes the interconnections between activities, namely their predecessors and successors. The collection of activities and links forms a directed acyclic graph. Each activity is assumed to be defined by two events: the start of the activity and the completion of the activity. The order in which activities are done depends on predecessor and successor relationships between the events and the activities.

Using the Gantt model or the CPM approach one may derive algorithms for computing certain functions over the project data. Given project start, project end, and activities' duration, algorithms can compute the following information for each activity: 1) earliest starting time, 2) lasting starting time, 3) earliest completion time, 4) latest completion time, 5) maximum available time, and 6) slack; see [11] for details.

The PERT Model: The Program Evaluation and Review Technique (or PERT) is related to the CPM approach. However, rather than a fixed start and end date for each activity, three time estimates are made for activity time: 1) the probable earliest completion time, 2) the probable latest completion, and 3) the most probable completion time. Time is measured by random variables with an assumed probability distribution. Thus the activity time is defined as a random variable with a beta distribution as shown in Fig. 3.

From the beta distribution one may derive the following equations:

$$\text{mean for activity time} = (a + 4m + b)/6$$

$$\text{variance for activity time} = [(b - a)/6]^2.$$

C. Inadequacies of the Existing Models

All of these three approaches, Gantt, CPM, and PERT, focus on the time scheduling of activities. The underlying graph representation makes the computation of useful scheduling functions straightforward. However, we have already observed that this simple formulation does not capture important characteristics of a design project, with the consequence that the schedule often goes out-of-date. Having looked at the traditional models, we can summarize their deficiencies:

- None of these models provides the information that would permit the manager to analyze and reason about the progress of activities.
- Current models are inadequate for representing the *wbs* as an integral system component.
- Although activity dependencies handle predecessor activity completion, they do not include the notion of boolean conditions.
- Current models are unable to regenerate and reschedule activities automatically.
- Current models are not capable of providing disjunctive dependency specification.
- Current models do not record the events that trigger the start of an activity.

III. Some Potentially Useful Models

In this section we provide a brief introduction to two computational models that we intend to use to describe

the software development process. Neither model by itself is adequate to describe this process. However, as we shall see, in combination they correct for all the deficiencies measured at the end of Sections I and II.

A. AND-OR Graphs

AND-OR graphs are used in artificial intelligence to model a task in terms of a series of goals and subgoals [21]. Each goal is represented by a node and its successor nodes are, in some sense, more primitive goals. Those goals, which can be satisfied only when all of their immediate subgoals are satisfied, are represented by AND nodes. Other goals, which can be satisfied when any of their immediate subgoals are satisfied, are represented by OR nodes. In many applications, the graph is usually generated at run-time while the program is attempting to satisfy the main goal. AND-OR graphs have also been proposed as a model for describing CAD/VLSI systems [14]. An *AND-OR graph* is a directed, acyclic graph containing three types of nodes:

1) An AND node denote an object that is an aggregation of all of its successor nodes.

2) An OR node denote an object defined by only one of its successor nodes.

3) LEAF nodes denote atomic entities. They have no outgoing arcs.

During its initiation, a software project is usually decomposed top-down until the software functional elements are sufficiently small to be estimated, planned, and executed. Establishing a *wbs* becomes the major task in the early stage of software development and a representation scheme is required. The *wbs* is a directed acyclic graph, so it makes sense to examine the AND-OR graph to see if the extension of AND/OR nodes produces additional modeling power. The traditional *wbs* essentially has only AND nodes. Adding the idea of an OR node would be an improvement, but a small one. The problem of multiple instances of the *wbs* is not solved, the problem of parallel activities is not addressed, and the problem of the activities changing radically is not addressed.

B. Petri Net Model

The Petri net model is an abstract model for describing and analyzing information and control flow in asynchronous concurrent systems [16]. The relationships between the parts of a system can be represented by a graph or network. The graph consists of two types of nodes: *places* (represented by circles) where one or more *tokens* (represented by small dots) can reside and the *transitions* (represented by bars) which can be *fired* to move tokens from inputs to outputs. A Petri net with tokens is called a marked Petri net and the number of tokens in a place is called the marking of that place. A transition is enabled when all of its input places have tokens. The firing of a transition causes tokens to be moved from their input places to their output places.

Fig. 4 shows the Petri net of a typical send-receive mechanism in a communication system. It models three

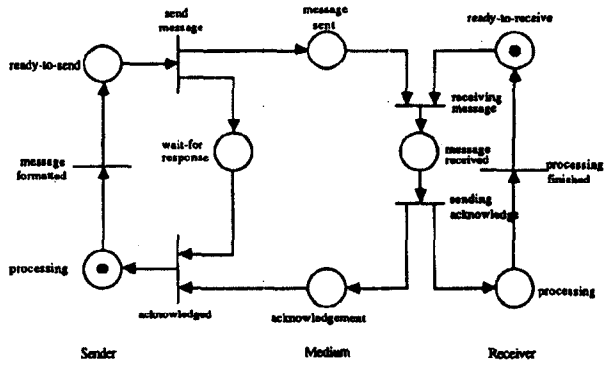

Fig. 4. Petri net of a sender-receiver system.

parts of the system: the sender, the receiver, and the medium. Initially, two tokens are placed in the net. This is called the initial marking of the Petri net. The sender is in processing state (i.e., formatting a message according to a designated protocol). The receiver is in a ready-to-receive state waiting for any incoming message. Once a message is formatted (the new command formatted transition is fired), the sender will transit to a ready-to-send state. When the message is transmitted through the communication medium, the send-message transition is fired and two tokens are created, one to the wait-for-response place on the sender side, the other to the message-sent place on the medium. Now the receiving message transition on the receiver side can be fired and the receiver transits to message-received state. After the message is verified, receiver sends an acknowledgment response to the sender and starts to process this message. When this response is received by the sender side, the sender goes back to processing state and is ready to accept any further request. When the processing of the message is finished, the receiver also goes back to ready-to-receive state.

In this example, static properties of a system are modeled by the graphical representation of a Petri net. Dynamic properties of a system result from its execution and can be determined by the net graph, the initial marking, and the transition firing rules. The net graph models two aspects of systems: events and conditions, and their relationships [16]. Tokens in places denote the existence of certain conditions. The fact that these conditions hold may cause the occurrence of certain events. Firing a transition can be considered as the happening of an event. It may change the state of the system, causing some conditions to cease holding and others to begin to hold.

Formally, a Petri net C is defined as a four-tuple $C = (P, T, I, O)$. P (the set of places) and T (the set of transitions) are the two major components of a net. They are associated with the circles and bars in a Petri net graph. Directed arcs from the places to the transitions and from the transitions to the places are represented by the input and output functions. The input function I defines, for each transition t_j, the set of input places for the transition $I(t_j)$. The output function O defines, for each transition t_j, the set of output places for the transition $O(t_j)$. A

marking vector $\mu = (\mu_1, \mu_2, \cdots, \mu_n)$ gives the number of tokens in a place for each place at a particular time. The number of tokens in place p_i is μ_i. A Petri net $C = (P, T, I, O)$ with a marking μ becomes the marked Petri net, $M = (P, T, I, O, \mu)$.

The next-state function δ defines the change in state by firing a transition. It changes the marking of the Petri net μ, to a new marking μ'. If t_i is enabled in marking μ, then $\delta(\mu, t_j) = \mu'$, where μ' is the marking that results from removing tokens from the input of t_j and adding tokens to the outputs of t_j. Two sequences result from the execution of the Petri net: a sequence of markings $(\mu^0, \mu^1, \mu^2, \cdots)$ and a sequence of transitions $(t_{j(0)}, t_{j(1)}, t_{j(2)}, \cdots)$ such that $\delta(\mu^k, t_{j(k)}) = \mu^{k+1}$ for $k = 0, 1, 2, \cdots$.

Since the Petri net model is a more powerful model than finite-state machines, many analysis questions turn out to be unsolvable or solvable at a high cost requiring exponential time and space. Several subclasses of Petri net pose restrictions on its modeling ability in the hope that the decision power will be increased [16]. There are also several extensions of Petri nets proposed that either increase the modeling power or adapt to certain types of applications. Some extensions add a predicate to transitions and associate attributes with tokens to represent value-dependent semantics. In [22], a modified Petri net model is used to describe a distributed software system design representation. A method [15] for translating Petri nets into a special procedural language named XL/1 which can, in turn, be optimized and translated into existing compiled languages has also been presented.

Another extension to the classic Petri net definition is to incorporate the notion of time. In the extended model of [17], transitions are used to model processes and the firing of a transition becomes an event with a duration equal to the execution time of the process. A similar extension was made by [3] with the exception that places are used to represent processes. The execution of a process is modeled by a transition representing the instantaneous start of execution with a directed arc to a place representing the condition of that process being in execution.

In summary, there are several advantages of modeling a concurrent system using Petri nets. Its graphical representation provides an intuitive and informal view of the underlying system behavior. The existence of analysis techniques allows the designer to derive the properties of a system and determine the complexity to verify whether the modeled system has such properties. The ability to model a system hierarchically facilitate the designer to apply top-down or bottom-up approaches during the design phase. It is especially suitable for monitoring progress of concurrent activities with an extension of the time domain as in [3]. This gives rise to further investigation of its potential application in modeling the software design process.

C. Inadequacies of These Models

The hierarchical nature of the AND/OR graph model provides a structural view into the modeled design objects. However, it only describes the vertical structure. The information involved in the software process consists of several types, namely, the product—both the internal documentation and deliverable end times, the activity—carried out to accomplish the project, the resource—consumed during the execution of the project, and the status report—issued at various checkpoints. The AND/OR graph model does not provide the dependency relationships among different information types. Since for each information type, an associated *wbs* can be constructed with it, we have a set of *wbs*'s represented in AND/OR graphs. Without further structural connection, we cannot know what resources are utilized during the execution of an activity to generate the required product. There is also no traceability among information in different phases. Several managerial questions, such as: what is the associated specification for a specific code module, and what responsible staff need to meet if a portion of the requirement is not met, cannot be answered.

If one tries to apply the Petri net model directly to software project management, this is also inadequate in several aspects. Suppose a place is used to represent a planned activity and a token inside it means that this activity is currently active (being executed). Related information, such as when it is started, how soon it is expected to finish, and how much it contributes to the overall project cost, must be attached to this token. But, a token only assumes a boolean value condition. Associating properties with tokens enables the net to carry more data flow information. Thus, nodes or places in a Petri net need to be of several different types. In our proposed model, a token is defined as a composite object that carries more information.

Second, the execution of a Petri net is nondeterministic. If more than one transition is enabled at any time, it is not predictable which one will fire first. This makes it more complicated to analyze the properties of a system. In an actual implementation, we can associate an executable procedure with each transition. The procedure will be invoked whenever the transition is enabled and determine globally what transition to fire first if more than one are enabled. This approach is explained in the next section. The nondeterminism is eliminated via this mechanism.

Furthermore, the Petri net model does not provide for the representation of aggregates of nodes. In a knowledge representation scheme, some properties of higher level nodes are defined through aggregate functions over properties of lower level nodes. For instance, the cost of a task is a summation of the cost of its constituent activities and the completion time of a task is the latest completion time among its constituent activities. This capability should not be confused with the hierarchical modeling in a Petri net. A Petri net model allows an entire net to be replaced by a single place or transition for modeling at a more abstract level. It also allows places and transitions to be replaced by subnets to provide more detailed modeling [16]. Thus it cannot be directly used for modeling the *wbs*.

To keep track of the project execution history for later reasoning, the token flow through the net must be recorded. The transition firing in a Petri net is volatile. This means once a transition is fired, the tokens in its input places are removed. There is no way to inquire if a token had existed earlier at a place. It does not provide for multiple instances of tokens in the net. Thus it cannot be directly used for modeling the event that occurs when an already executed activity now needs to be re-executed.

IV. A Hybrid Model for Software Project Management

In the first subsection we present the formal definition of DesignNet.[2] In the second subsection we show how the waterfall model and a complete *wbs* is mapped onto a DesignNet. The third subsection shows how the model supports iterative process such as project replanning and project history. The final subsection formally defines the notions of connected, plan complete, plan consistent, and well-executed. It then gives efficient algorithms for their computation.

A. The DesignNet Model

Basically, the DesignNet model follows the terminology used in AND/OR graphs and Petri nets. We will give an introductory example and illustrate how a portion of a software project can be mapped onto this model. A formal definition is given after this brief introduction.

A DesignNet consists of a set of places, a set of structural operators, and a set of transitions. Places are *typed*. A token of a specific place type can only represent information of that type. Four place types are defined including *activity*, *resource*, *product*, and *status report*. Structural operators connect places of the same type on two adjacent levels where the lower level places are the decomposition of the higher level place. The hierarchy resulting from the connection of structural operators is the *wbs* of the project. Prerequisite conditions (products and resources required) before an activity can start and the products generated by an activity are linked to activities by transitions. This linkage defines the dependencies among life cycle phases. The firing of transitions creates new instances of tokens and simulates the project execution process. Each token is associated with time dependent information and represents an occurrence of an iteration in the software development process. For example, tokens in a product place denote various versions of that product.

An example of a DesignNet is shown in Fig. 5. The five activity places *activity*, *design & implementation*, *testing*, *driver design & imple.*, and *driver testing* together with the AND operator firm an activity *wbs*. The product places *graphic command spec.*, *test plan & procedure*, *driver code*, *operation manual*, *baseline code*, and *version description*, also form a product *wbs*. The *driver design & implementation* activity is initiated either when a new version of the graphic specification is generated or when the testing shows a failure. Both situations require the resource, *system programmer*, to be involved in the design task. Other elements in this figure, such as the concepts of place type, instantiated tokens, aggregation with structural operators, and transition firing rules, will be explained in the following paragraphs.

Formal Definition: DesignNets are composed of three basic components:

- A set of *places* P, where P is a union of four possible *place types*: P_a for activities, P_r for resources, P_p for products, and P_s for status report. Specific symbols are used to represent these place types. An oval represents an activity place. A punched card symbol is used for the resource place, since resources can be considered as input of activities. Product places are drawn as printed output symbols as in conventional flowcharts, since they are entities generated after activities are finished. A square box represents a status report place.
- A set of *structure operators* S is a union of two types of operators, an AND operator, S_a, and an OR operator S_o, where the operator connects nodes of the same type, (activity to activity, product to product, resource to resource), and the AND operator defines an AND relationship among the successor nodes and the OR operator defines an OR relationship among the successor nodes. Graphically, an AND structure operator is represented by a boolean logic AND gate symbol, and an OR structure operator is represented by a boolean logic OR gate symbol.
- A set of *execution transitions* T, where T is a union of two possibilities, T_s, T_f where each is an executable procedure. This makes the model deterministic. T_s is a transition to start an activity and T_f is a transition after an activity is completed. In graphical representation, a bar represents a transition. Since it is clear from the position of the bar whether it stands for a T_s or a T_f transition, no visual distinction is made between T_s and T_f.

Places are strongly-typed. Each place type can be further classified into several subtypes. For example, the requirement document, preliminary design specification, interface design specification, code, test plan, and operating instruction manual are possible subtypes of the product place type. The *tokens* resident in places carry more information than just a boolean value. For each place type, a set of properties are defined and any token in a place becomes a composite object of a specific *token type*. For example, a token in a resource place means that a resource has been allocated in anticipation of an activity starting. In addition to its allocation status, the following relevant information can be stored within this token: resource type (system analyst, programmer, or technical writer), skills, home department, unit cost, percentage of working time available, etc.

Places of the same type form a hierarchical lattice via the connection of structural operators. There is one place

[2]This name is based on its applicability to a design oriented project. It should not be confused with only the design phase of the software development process.

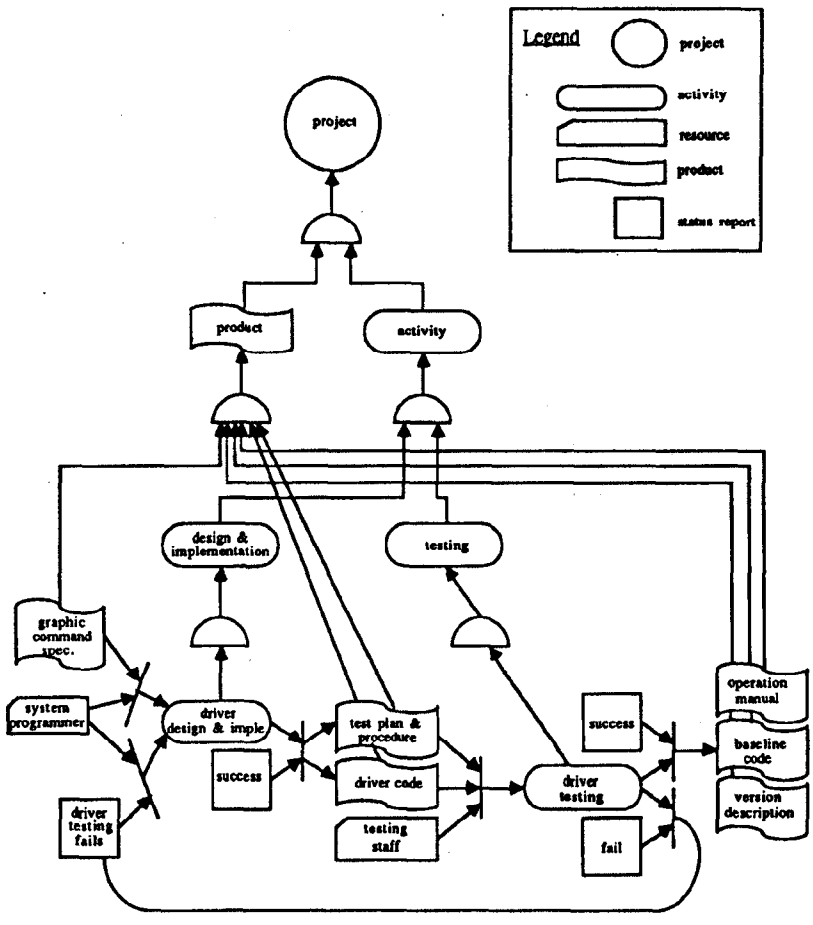

Fig. 5. An example of the DesignNet.

node denoted as the root and defined to be at level zero. The AND operator defines the subcomponents of an aggregate entity. The OR operator specifies what alternatives are available for a design object. The manner in which places on adjacent levels are connected is defined by the four functions: I_a, I_o, O_a, O_o.

- The function I_a represents the set of input arcs for connecting a place node to an AND operator S_a.
- The function I_o represents the set of input arcs for connecting a place node to an OR operator S_o.
- The function O_a represents the set of output arcs from an AND operator to a place one level above the place associated with its input.
- The function O_o represents the set of output arcs from an OR operator to a place one level above the place associated with its input.
- In the graph representation, the four functions I_a, I_o, O_a, O_o are represented by the directed arcs from the places to the operators and from the operators to the places, respectively.

The structural operators have two major functions. First, they allow aggregate information of a higher level place to be collected from its constituents. For example, we can define the cost of a higher level activity as a summation of the cost from its AND children activities and the completion time of a higher level activity as the latest completion time among its AND children activities. Second, they allow newly created tokens on lower levels to propagate upward. Let us use an example to illustrate how this mechanism works. Suppose we have a product of program module p_1 which is further decomposed into two submodules p_2 and p_3 as shown in Fig. 6(a). At time i, a version $p_{2(i)}$ of p_2 is generated. The AND operator will automatically create a token instance $p_{1(i)}$ of p_1. This is shown in Fig. 6(b). At time j, a version $p_{3(j)}$ of p_3 is generated. The AND operator now creates an instance $p_{1(j)}$ which is composed of $p_{2(i)}$ and $p_{3(j)}$ as Fig. 6(c). Later, at time k, due to the existence of a new version of p_2, a new version $p_{2(k)}$ is generated. Automatically a token $p_{1(k)}$, which is composed of $p_{2(k)}$ and $p_{3(j)}$, is created as in Fig. 6(d). This mechanism provides valuable assistance for version control and configuration management.

The relationships among different place types are defined by the connections of transitions. The manner in which the places on the same level are connected is defined by four functions I_s, I_f, O_s, O_f.

- The function I_s represents the set of input arcs for a transition T_s, from a place which is of type product or resource.
- The function I_f represents the set of input arcs for a

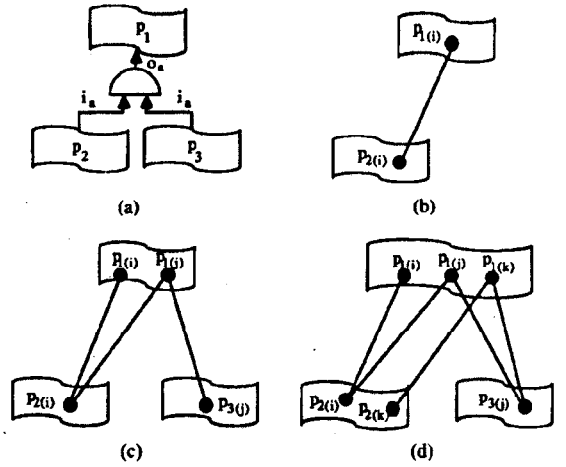

Fig. 6. The token propagation through structural operators.

transition T_f, from a place which is of type activity or status.

- The function O_s represents the set of output arcs for a transition T_s, to a place which is of type activity.
- The function O_f represents the set of output arcs for a transition T_f, to a place which is of type product.
- In the graph representation, the four functions I_f, I_s, O_f, O_s are represented by the directed arcs from the places to the transitions and from the transitions to the places, respectively.

Note that an activity place is never connected to another activity place with a transition between them, since T_s only connects product and resource to activity and T_f only connects activity to product. This restriction avoids establishing the dependencies directly between activities and removes the limitation of the traditional PERT approach that only allows conjunctive "AND" dependent conditions to be specified. In Fig. 5, the plotter driver design activity will be triggered either when a new version of graphic specification is generated or when the testing shows a failure. Thus, disjunctive dependencies can be built.

A *DesignNet graph* D is defined as a five-tuple $D = (P, T, S, I, O)$ where $P = \{P_a, P_r, P_p, P_s\}$, $T = \{T_s, T_f\}$, $S = \{S_a, S_o\}$, $I = \{I_s, I_f, I_a, I_o\}$ and $O = \{O_s, O_f, O_a, O_o\}$. Fig. 7 summarizes the symbols used and their connections.

The DesignNet graph still preserves the *bipartite* property satisfied by Petri nets, since its nodes can be partitioned into three sets (transitions, places, and structural operators) and in the two subset combinations $(P, S, I_a, I_o, O_a, O_o)$ and $(P, T, I_s, I_f, O_s, O_f)$ each arc is directed from an element of one set to an element on the other set. Notice that there is no arc linkage between transitions and structural operators, the two sets S, T are disjoint. Another interesting property is the projection of the net into subsets of its domain. Removing T and its connections I_s, I_f, O_s, O_f, we get four partitions (activity, resource, product, status report) of the *wbs* and lose the dependencies between different information types. Furthermore, if we want to focus on only one product type, for example the

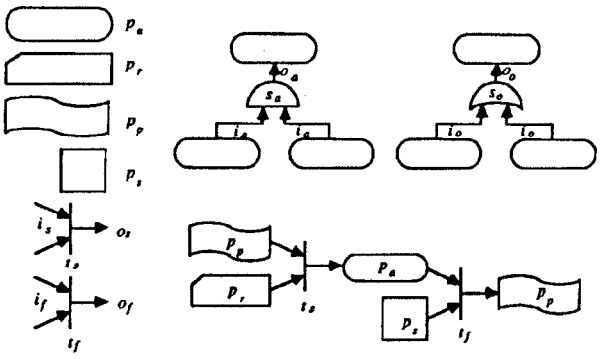

Fig. 7. DesignNet graphical representation.

program, we can further prune away the undesired product types and have a program configuration tree resulting. Removing S and its connections I_a, I_o, O_a, O_o, the bottom level of the net becomes an activity network analogous to a PERT chart, but the hierarchical *wbs* is lost. Thus, the projection of a DesignNet into various domains provides different views of a project.

A DesignNet executes by *firing* transitions. Unlike the firing of a transition in the traditional Petri net, which causes tokens to be moved from their input places to their output places, the transition firing in a DesignNet is a *nonvolatile* process. A token can be in one of three possible states, *active*, *consumed*, or *discarded*. When a token was created, it was put in an active state. A transition may fire if it is *enabled*. A transition is enabled if each of its input places has at least one active token in it. Firing a transition does not remove tokens from its input places. Instead, it changes the tokens in its input places to a consumed state such that any further firing resulting from the same token is prohibited. The tokens created in its output places are put in an active state. While creating new tokens, it also checks whether any token in output places is already in an active state. If there are, then it changes them to a discarded state. This guarantees that at most one token is in an active state for any place at any time.

Note that we permit any number of tokens to reside at a place, and each token would represent an instance of the place when the token was created. Depending on place type, the number of tokens inside a place has different meanings. The number of tokens in an activity place means that the activity has been executed this number of times. If we want to know the detail of each execution, we can inquire for the information associated with an individual activity instance. On the other hand, the number of tokens in a product place means that there are so many versions generated with this product. Each token has a pointer to the data it represents (e.g., the file name of a code module). Since a token is not removed (just consumed) after a transition is fired, the number of tokens in a place will increase monotonically.

To handle the transition firing operation, each transition can be considered as an executable procedure. When a new token is created in the input places of a transition, the procedure associated with the transition will be exe-

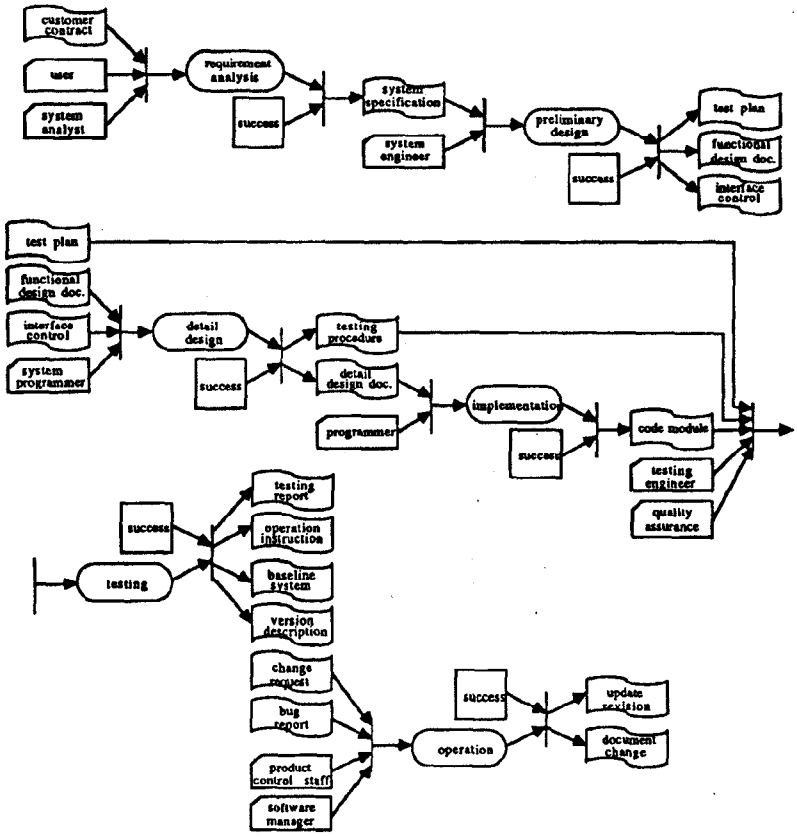

Fig. 8. DesignNet of life cycle phases.

cuted. This procedure would check against all the input places. If all the input places have active tokens, it performs the *firing* operation by setting the input tokens to consumed state and creates new tokens in output places through instantiation. This mechanism can be achieved by a triggering operation using object programming techniques.

A marking μ of a DesignNet is an assignment of *tokens* to the places in that net. The vector $\mu = (\mu_1, \mu_2, \cdots, \mu_n)$ gives the number of tokens in a place for each place in the DesignNet and the number of tokens in place p_i is μ_i, for $i = 1, \cdots, n$. Since the transition firing is nonvolatile, the number of tokens in a place will never decrease and for a place p_i, $\mu_{i(t)}$ is greater than or equal to $\mu_{i(s)}$ for any time t later than the time s. The sequence of markings $(\mu^0, \mu^1, \mu^2, \cdots)$ at the time index $0, 1, 2, \cdots$, together with the information stored with individual tokens, gives the execution history of a DesignNet.

With all the previous definitions, a *project* J can be defined as a seven-tuple $J = (R, P, T, S, I, O, \mu)$ where P, T, S, I, O, μ are defined as before and R is the root place representing the project as a whole. The level next to R, connected through an AND structural operator, contains the four activity, resource, product, and status report *wbs*.

B. Mapping the Waterfall Model onto the DesignNet

The waterfall model [19] for software development contains six major phases. Different personnel skills and development disciplines are required for different phases. Also, each phase has its own product types. Fig. 8 shows the DesignNet representation of the six development phases.

This life cycle phase representation works as the top level of the *wbs*. The project manager determines what methodology is used for each phase and defines the structure and acceptance criteria of the end product. Verifying functions of these accepting criteria can then be attached to the finish transition of predefined activity types. In the initial *requirement analysis* phase, the user and the system analyst are involved in defining the system specification based on the customer contract. Tools for automating the requirement and specification development, such as SREM[5], can be used at this stage. During *preliminary design*, system engineers take the specification as input and generate the functional design document, the interface control document and the test plan of each module. Special design tools (e.g., PSA/PSL [20]) can be used at this stage. In *detail design* and *implementation*, programmers translate these design descriptions into code modules. The code modules together with test plans and testing procedures generated in earlier stages are then tested (*testing* phase) by testing engineers and quality assurance staff. When the final product goes into *operation*, product control staff and the software manager are in charge of the update and document change.

For the sake of simplicity, this figure only shows the

transitions which are fired when activities are successfully executed. If any failure report is issued while an activity is being executed, proper backtracking transitions to previous phases are necessary. Such iteration paths are omitted in this figure. However, Fig. 5 shows one example of this repetition (from driver testing to driver design). In actual cases, many repetitions will occur and the project manager can add transitions with appropriate status report places that loop back to earlier phases.

C. Implications of the Model

This section describes how DesignNet can be used to facilitate important steps of the software development process.

Project Planning and Cost Estimation: At the beginning of a software project, the project manager is responsible for choosing the development methodology and defining the typical activity behavior and requirements of each phase. These definitions are then represented in the DesignNet model and entered into the system. The DesignNet of the waterfall model described in the previous section is just one template for developing the project plan. During the planning stage, each phase is decomposed into lower level subactivities that inherit the resoure requirements and product specifications from its parent activity with some elaboration. This decomposition is performed until a level is reached such that a concrete working package can be assigned. Fig. 9 shows a sample decomposition of the plotter driver design task.

After the *wbs* has been constructed, the project manager can invoke the cost estimation function to compute total project cost bottom-up, level-by-level. In the DesignNet, a user can attach different heuristic functions to evaluate properties of aggregate objects. Based on the methodology used, a design activity may use the number of person hours for estimation and an implementation activity may use number of lines-of-code for estimation. Actually, we only need to apply estimation functions to bottom level activities. The cost of intermediate level nodes are summations of constituent activities' cost and can be computed automatically. The existence of a *wbs* greatly reduces the manual effort spent on cost computation.

Project Network Construction: The *wbs* of each phase may be derived by individual phase supervisors. Somehow these phases must be combined to produce a single *wbs*. Using our model we can connect all the phases into a single *wbs* automatically by applying searching operators to the product places. Since an activity takes products as input, it can be started only after the activity that generates the required product is finished. The predecessor and successor relationships among activities are based on the document (or product) flow across phases. In contrast to traditional CPM and PERT approaches, where the dependencies are built directly upon activities and most of the planning efforts are spent on manually constructing the network, including product information in the project hierarchy makes automatic network construction possible.

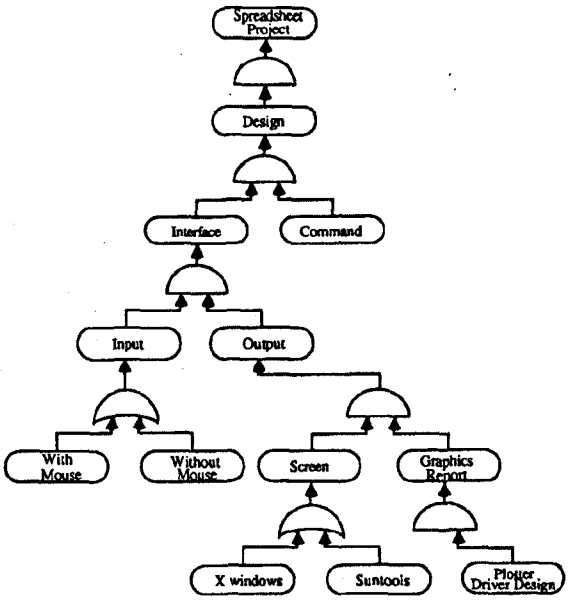

Fig. 9. Sample decomposition of a design task.

After connecting all the output product places to the input places of associated activities in the *wbs*, the project network is constructed automatically and project planning is then completed. In the meantime, the project manager can also discover incomplete or redundant parts of the plan, such as a specification which is not followed by a design activity, or a test plan that is not used by any testing activity.

Iteration and Automatic Activity Triggering: Due to the nature of a design project, the same[3] activity may be executed multiple times depending on various conditions. To avoid destorying the project history, different token occurrences of the same activity are created whenever it is executed. The firing of a start transition of an activity performs the instantiation operation. Since the activity only acts as a template, every time its preconditions are met, the start transition will be fired and create an instance of the token in the activity place and store the relevant information with this token.

During the execution of a project, the current project state is affected only when any token value is changed. The project progress can be considered as a token driven process. Unlike the Petri net which is a *closed* system, a DesignNet is an *open* system. All the status report places and certain product subtype places (e.g., customer contract, change request, and the bug report) accept externally supplied tokens. Responsible staff will submit to the system the tokens that reflect events in the process. The system will react according to the DesignNet definition.

Traceability: Effective management of numerous project information is a crucial factor to the success of a software project. The information includes activities exe-

[3]Here the "same" activity means the planned activity which may be executed at different times. Chronologically, two activities executed at different times cannot be the same even though they perform the same job.

cuted, resources consumed, reports issued, documents generated, plus the relationships between these entities and linkages across phases. Furthermore, unpredictable backtracking makes it more difficult to maintain these relationships. DesignNet provides an automatic information tracking mechanism. With the project history information being recorded automatically, the project manager can issue queries against the project database to analyze and reason about the project's progress. Most importantly he may query over data that is distributed in several phases. For instance, to know what code modules are affected with respect to a specification change, the model can start from the token associated with that change, trace through all the design, implementation and testing activities triggered, and then locate the resulting modules.

By tracing through the tokens and links, the model can derive the answer to several managerial questions. For example, what activities cause the major delay or cost overrun? If an activity has been executed often, who has been responsible for it? If we change the requirement for the plotter output, how much more time and resource will be needed to complete the job? What other activities are delayed? If the project shows some delay, is it possible to make up the delay by introducing extra resources? If yes, by adding resources to what activities? When personnel turnover occurs, can we find the staff who were involved with an activity executed two months ago?

Complete historical data is the basis for the satisfaction of an audit. DesignNet also provides the version development traceability at a coarse level of granularity. Tokens in the same place denote different occurrences of that place. Time dependent properties stored with each token provide flexibility for building finer level version control. For example, a version index is stored with the token of a code product place. We can incorporate any source code version control system (e.g., SCCS [18]) and use the version index to locate the version generated at a specific time.

Points of Clarification:

- The model is intentionally designed to be passive, in the sense that it permits the collection of data and it can report the complete history of the project, but it does not try to reallocate resources or even to determine when an activity has been successful or not. All of these latter issues are determined by the project manager or managers. We do not believe that at this time we know enough about building tools that automatically determine the allocation of resources or the definition of activities to satisfy a project.
- One should not view DesignNet as an extension of Petri Net, because there are major differences. In particular DesignNet is an OPEN system, and some tokens for certain places are assumed to be provided by project personnel. Another major difference is that DesignNet is deterministic because one needs in a software project to determine whether an activity succeeded or needs backtracking to previous activities.
- To represent the *wbs*, the choice was made to use AND/OR notation as opposed to transition diagrams. The major reason is because when a lower level place produces a new token (i.e., the state of the project has been changed), then that token must immediately be propagated up the *wbs*. The transition diagram notation does not fire until all dependent nodes are ready. The more accurate representation of the life cycle is through the use of AND/OR nodes.
- One must be careful about interpreting the AND and OR nodes in a DesignNet. When an OR node is used to connect several subactivities to an activity, the it means that when only one of those subactivities is done, then the parent activity is partially done. For example using Fig. 9, when the X-Windows activity is done, and all its parent activities to the root of the *wbs*, then even though the *wbs* is not completed, it is partially completed and the version of the software that supports X-Windows can be shipped. In fact, it may be the case that the root node of the *wbs* never reaches a state of completion, but is always partially complete and software is being shipped, but not all activities are finished executing.
- Another point is that the *wbs* does not necessarily reflect the structure of the code. It merely describes the way in which the activities are divided and the way in which products are decomposed.
- No plan can ever cover all of the possibilities. For example, suppose someone quits in the middle of an activity. How does DesignNet handle this? The manager would go into DesignNet and add a Status Report Node to the activity that has been aborted. He then reports what has occurred and adds in a transition to an activity that is to be followed at this point. That activity may be to assign a new resource and restart the previous activity again, or he may decide not to return to this aborted activity. Thus the project plan can still be updated to cover the unanticipated case.
- DesignNet does not explicitly handle resource allocation. In general people are working on multiple projects at the same time. The problem of optimal allocation is computationally intensive, and the factors that must be used to determine allocation are complex and not describable by a network model such as DesignNet. Therefore we assume that project personnel assign resources to activities.
- Although the example in this paper shows the waterfall model, it is equally easy to incorporate other life cycle development models into DesignNet.
- Consider the situation when driver testing uncovers bugs and driver design is restarted. Driver design completes and the driver is being recoded, when a change in specification causes driver design to be restarted. How does DesignNet detect the case that more than one of the same activity is concurrently active, and how does it resolve this anomaly? Detection in DesignNet is straightforward, as one can trace from an active node to the finished product to see if other active nodes exist. However,

the model assumes that project personnel will decide whether to abort one of the activities, or to continue with both of them.

D. Using DesignNet to Analyze Project Properties

Another application of the DesignNet is to take the graphical representation of a project, and analyze it for the presence of desirable or undesirable properties. With the properties derived from a DesignNet, we can derive the properties of the project which the net models. A project manager can use these analysis techniques to help him detect any problem concerning project planning or execution. This section defines several essential properties of the DesignNet and the algorithms to verify the existence of these properties are also presented.

Definition: A DesignNet is *connected* if and only if for all the places p, there exists a path from p to the root place R through structural operators and the arcs I_a, I_o, O_a, O_o.

This property defines project plan connectivity. For a large project with hundreds of activities, the planning is usually conducted by several task supervisors concurrently and each one builds only a partial *wbs* of the whole plan. This property ensures that, after integration, all activities contribute toward the accomplishment of the project and no irrelevant activity or product is created. To verify that the DesignNet of a project is well-formed, apply the following check for every place p_i, $i = 1,2,\cdots,n$. If p_i has no outgoing structural arc and p_i is not the root then stop and the net is not well-formed. Otherwise, for each outgoing structural arc (an element of I_a or I_o), check if it is directed to a structural operator and there exists an arc (an element of O_a and O_o) from this operator to another place.

Definition: A project is *plan complete* if and only if its associated DesignNet is connected and for all *intermediate* product places p_p, there exists a transition t_s in T_s such that t_s takes p_i as input and there exists a path from t_s to some final deliverable product places.

During the software development process, many types of documents are generated. Some of them are final deliverable products, such as release notes, executable code, and operational manual. Others are just for internal use, such as system requirement, interface design specification, control design specification, and test plan. We consider the products that are taken as input to some activity in successive phases as intermediate products. If no activity takes an intermediate product as input for further processing, part of the desired functions will not be implemented and the final developed system will be incomplete. This property assures the robustness of a plan. To check *plan completeness* of a project, apply the following procedure for each final deliverable product place: mark it as visited, find the activity place that generates this product, then mark all precedent product places of this activity as visited and recursively apply the same checking to these places. Upon completion, if there still exists any unmarked product place, the associated project is plan incomplete.

Definition: A project is *plan consistent* if and only if it is plan complete and for each activity place p_a, the *level* (distance to the root place) of this place is the same as the levels of its preceding and succeeding activity places.

A rigid hierarchical decomposition requires that at any level in the hierarchy, each subdivision must be consistent with other corresponding phases. For example, if there are three subtasks defined under the specification phase, the design phase has to be decomposed into three subtasks and each subtask performs its specific system design function in correspondence to the subtasks of the previous phase. If one of the specification subtasks is further decomposed into several subactivities, the associated design subtask must also be decomposed into the same number of subactivities to cover all aspects at the same level of detail. To check the plan consistency of a project, first compute the level of each activity place and then apply a similar search algorithm used to check plan completeness. This time, when tracing activity dependencies, make sure that the level of each activity along the path is the same.

Definition: A project is *well-executed* if and only if there is at most one token in an active state in all places at any time.

Several possibilities may cause a place to have more than one token in active state. Consider the plotter driver design example. If the drive design activity is active and a system analyst decides to add in new graphic commands, a new design activity will be initiated in response to this change. Even though this kind of late modification is discouraged by software design methodology, such situations do happen regularly in actual practice. However, if we allow the same activity to be executed by different personnel based on different versions of input documents at the same time, it will result in generating consistent products. One way to resolve this anomaly is to cancel the current active activity when it needs to be re-executed. Meanwhile, we have to release the allocated resources, reschedule the successive activities, and call a staff meeting if different personnel are assigned to these two activities. Transition firing in DesignNet changes a token from *active* state to *discarded* state when it creates a new token of the same place. This mechanism enforces a project's well-executed property. Since the user can attach executable procedures to a transition in a DesignNet, the resource release, rescheduling, and staff synchronizaton can be fulfilled by these procedures.

V. Epilogue

Software engineering research has successfully focused on tools to aid individual life cycle activities, such as requirements analysis or design. Far less effort has been spent considering how to support the managers of the software development process. This is surprising, as so much of what makes large scale software development distinctive is the large managerial component involved.

Recent work on integrated CASE environments attempts to place all "useful" information in a common database. This has many advantages, one being the ability to query over different artifacts of the process and to check consistency between them. The DesignNet model attempts to bridge the gap between the underlying representation of data and the interface to the user. Without such a model, the user is faced with a vast amount of information, but little organization. We feel that a model that incorporates the *wbs* and task reinitiation is essential for effectively organizing and tracking the project. We believe that consistency of the data needs a framework upon which one can base algorithms that will work with relative efficiency. Both of these are provided by DesignNet.

DesignNet is a model for describing and monitoring the software development process. It utilizes AND/OR structure operators to describe the work breakdown structure and Petri net notation to represent the dependencies and parallelism among activities, resources, and products. Tokens are objects with specific properties. Token propagation through structural links allows aggregate information to be collected automatically at different levels of detail. The transition firing is a nonvolatile process and creates new token instances with time dependent information. The typed places, together with connections among them, defines the static construct of a project. Whenever transitions are fired, the project execution history is recorded by the token instances created.

Using the model, we have provided definitions for basic properties of a successful project, namely connectedness, plan complete, plan consistent, and well-executed. We have given algorithms for computing these functions and shown that the computing time is linear in the size of the project. This assures that any system based on DesignNet should be able to compute these functions efficiently. Finally, we have shown how the waterfall life cycle model maps onto a DesignNet and the implications for project planning, cost estimation, project network construction, reinitiation of activities, and traceability across the life cycle. Other life cycle models can be equally treated.

Large software development projects can take many forms and exhibit myriad patterns of behavior. It is therefore unrealistic to believe that some simple underlying conceptual model exists that will explain it all. To make such a model complete, in the sense that it does handle all of the possibilities that may occur, a powerful yet potentially complicated model is inevitable. DesignNet inherits the power of the Petri net model, and includes extensions that make it suitable for describing the software life cycle. We intend to, and we hope that others will also, develop a software project management system that uses DesignNet as its base.

Currently, we are building a prototype system, called DesignPlan, based upon the proposed model. The underlying database support for the system is an object-oriented database (Vbase from Ontologic, Inc.) that provides all the object types and triggering mechanisms definition. Three functional modules are implemented on top of the DesignNet conceptual layer—the activity scheduling module, the resource allocation module, and the cost estimation module. The front end is an X window based graphic user interface. Users can browse or update the project information through an object navigator. Its result and detail design information will be reported in a later paper.

Acknowledgment

We would like to thank the referees for their helpful suggestions.

References

[1] W. W. Agresti, "What are the new paradigms," in *New Paradigms for Software Development: Tutorial*. Washington, DC: IEEE Computer Society, 1986, pp. 6–10.

[2] B. W. Boehm, *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall, 1981; see also *IEEE Trans. Software Eng.*, Jan. 1984.

[3] J. E. Coolahan, Jr., and N. Roussopoulos, "Timing requirements for time-driven systems using augmented Petri nets," *IEEE Trans. Software Eng.*, vol. SE-9, no. 5, pp. 603–616, Sept 1983.

[4] B. Curtis, H. Krasner, V. Shen, and N. Iscoe, "On building software process models under the lamppost," in *Proc. 9th Int. Conf. Software Engineering*, Monterey, CA, Mar. 30–Apr. 2, 1987, pp. 96–103.

[5] C. G. Davis and C. R. Vick, "The software development system," *IEEE Trans. Software Eng.*, vol. SE-3, no. 1, pp. 69–84, Jan. 1977.

[6] E. W. Davis, Ed., *Project Management: Techniques, Applications, and Managerial Issues*. Industrial Engineering and Management Press, 1983.

[7] M. Dowson, "Iteration in the software process," in *Proc. 9th Int. Conf. Software Eng.*, Monterey, CA, Mar. 30–Apr. 2, 1987, pp. 36–39.

[8] W. J. Fabrycky, P. M. Charge, and P. E. Torgersen, *Applied Operations Research and Management Science*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

[9] E. Horowitz, and R. C. Williamson, "SODOS: A software documentation support environment—Its definition," *IEEE Trans. Software Eng.*, vol. SE-12, no. 8, pp. 849–859, Aug. 1986.

[10] ——, "SODOS: A software documentation support environment—Its use," *IEEE Trans. Software Eng.*, vol. SE-12, no. 11, pp. 1076–1087, Nov. 1986.

[11] E. Horowitz and S. Sahni, *Fundamentals of Data Structures in Pascal*, 2nd ed. Rockville, MD: Computer Science Press, 1985.

[12] H. Kerzner, *Project Management, A Systems Approach to Planning, Scheduling, and Controlling*. New York: Van Nostrand Reinhold, 1984.

[13] C. C. McCracken and M. A. Jackson, "A minority dissenting opinion," in *Systems Analysis and Design—A Foundation for the 1980s*, W. Cotterman, Ed. New York: Elsevier, pp. 551–553.

[14] D. McLeod, K. Narayanaswamy, and K. V. Bapa Rao, "An approach to information management for CAD/VLSI applications", in *Proc. ACM-SIGMOD Int. Conf. Management of Data*, May 1983.

[15] R. A. Nelson, L. M. Habit, and P. B. Sheridan, "Casting Petri nets into programs," *IEEE Trans. Software Eng.*, vol. SE-9, no. 5, pp. 590–602, Sept. 1983.

[16] J. L. Peterson, "Petri nets," *ACM Comput. Surveys*, Sept. 1977.

[17] C. V. Ramamoorthy and G. S. Ho, "Performance evaluation of asynchronous concurrent systems using Petri nets," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 440–449, Sept. 1980.

[18] M. J. Rochkind, "The source code control system," in *Proc. First Nat. Conf. Software Engineering*, IEEE, New York, 1975, pp.37–43.

[19] W. W. Royce, "Managing the development of large software systems," in *Proc. 9th Int. Conf. Software Engineering*, Monterey, CA, Mar. 30–Apr. 2, 1987, pp. 328–338.

[20] D. Teichroew and E. A. Hershey, "PSL/PSA: A computer-aided technique for structured documentation and analysis of information

processing systems," *IEEE Trans. Software Eng.*, vol. SE-3, no. 1, pp. 41-48, 1977.
[21] P. H. Winston, *Artificial Intelligence*, 2nd ed. Reading, MA: Addison-Wesley, 1984.
[22] S. S. Yau and M. U. Caglayan, "Distributed software system design representation using modified Petri nets," *IEEE Trans. Software Eng.*, vol. SE-9, no. 6, pp. 733-745, Nov. 1983.

Lung-Chun Liu received the B.S. degree in computer science from National Chiao-Tung University, Taiwan, Republic of China, in 1980, the M.S. degree in electrical engineering from National Taiwan University, Taiwan, Republic of China, in 1982, and the Ph.D. degree in computer science from the University of Southern California, Los Angeles, in the area of software project management, in 1988.

He is now with Cadence Design Systems, Inc., Santa Clara, CA. His principal research interests include: design and application of object-oriented databases, project management tools for software development, and integrated software engineering environments.

Ellis Horowitz received the Ph.D. degree in computer science from the University of Wisconsin.

He was on the faculty there and at Cornell before assuming his current post as Professor of Computer Science and Electrical Engineering at the University of Southern California, Los Angeles. He has published over 100 research articles on subjects ranging from algorithms, data structures, computer-assisted instruction, and software engineering. He is also the coauthor of several popular textbooks.