IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE, VOL. PAMI-7, NO. 5, SEPTEMBER 1985

Representation of Activity Knowledge for Project Management

ARVIND SATHI, MARK S. FOX, MEMBER, IEEE, AND MICHAEL GREENBERG

***Abstract*—Representation of activity knowledge is important to any application which must reason about activities such as new product management, factory scheduling, robot control, vehicle control, software engineering, and air traffic control. This paper provides an integration of the underlying theories needed for modeling activities. Using the domain of large computer design projects as an example, the semantics of activity modeling is described. While the past research in knowledge representation has discovered most of the underlying concepts, our attempt is toward their integration. This includes the epistemological concepts for erecting the required knowledge structure; the concepts of activity, state, goal, and manifestation for the adequate description of the plan and the progress; and the concepts of time and causality to infer the progression among the activities. We also address the issues which arise due to the integration of aggregation, time, and causality among activities and states.**

***Index Terms*—Activity, AI, causality, goal, knowledge representation, manifestation, time, truth propagation.**

I. Introduction

THE management of activities in large projects is composed of four parts.

1) Planning: Definition of activities and specification of precedence, resource requirements, durations, due dates, and milestones.

2) Scheduling: Selection of activities to perform (if more than one way exists), and the assignment of actual times and resources.

3) Chronicling: Monitoring of project performance, detection of deviations from the schedule, and the repair[1] of the original schedule (possibly resulting in renewed planning and scheduling).

4) Analysis: Evaluation of plans, schedules, and chronicled activities for normal reporting and the detection of extraordinary situations.

Central to the performance of these activities is the availability of a theory of activity representation. This would have to be comprised of two parts: syntactic conventions and a set of semantic primitives. It would h satisfy three criteria.

1) Completeness: represents all relevant con Given an application, completeness requires that th resentation span the domain.

2) Precision: provides appropriate granularit knowledge. The representation should be capable (scribing the domain situations at the level of precisior in the domain.

3) Clarity: lacks ambiguity in interpretation. ' domain languages are typically ambiguous, the repr tation should provide clarity by ensuring that eacl situation corresponds to one and only one model.

The importance of such a theory is crucial not or the construction of project management systems but t application which must reason about activities. The: clude factory scheduling, robot control, vehicle co software engineering, and air traffic control. This | provides the basic elements of the theory needed for eling activities which can be used for the knowledg gineering in such planning, scheduling, and/or pro chronicling tasks.

Considerable effort has gone into constructing piec such a theory, e.g., the aspects of time [1], causality activity [2], authority [27], constraint representation and ownership [21]. What is missing is a unificatic these ideas into a single theory and a test of its adeq

Since 1982, the Callisto project [34] has been structing such a theory in the context of engineering ect management. The role of project management hε creased in importance. Innovation is becoming cruci the continued vitality of industry. New products an novations to existing projects are occurring with inc ing rapidity while product lives decrease. In an effc maintain a market share, companies are forced to re project development time. By entering the market as (as possible, the product life may be extended. Produc velopment time may be reduced by product simplific: or through better management of the development ac ties. Our focus is on the latter.

Experience has shown that project management ha: come more difficult, especially in the high-technolog dustries. A close observation of project activities sh that errors and inefficiencies increase as the size of project grows. The successful performance of project t are hindered by the following.

- *Complexity:* due to the number and degree of ir actions among activities. For example, in a computer sign project, a design engineer's decision to use one

Manuscript received May 17, 1985. This work was supported by the Digital Equipment Corporation. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Digital Equipment Corporation.

A. Sathi was with the Intelligent Systems Laboratory, Robotics Institute, Carnegie-Mellon University, Pittsburgh, PA. He is now with Carnegie Group, Inc., Pittsburgh, PA 15219.

M. S. Fox is with the Intelligent Systems Laboratory, Robotics Institute, Carnegie-Mellon University, Pittsburgh, PA 15213.

M. Greenberg is with the Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003.

[1]Repair or debugging involves three activities: information collection/management, analysis, and replanning/rescheduling. Chronicling stands for the information collection and management aspects of repair, while analysis, planning, and scheduling are covered elsewhere.

ticular integrated circuit may affect the supply of parts and production of prototypes by the manufacturing people.

- *Uncertainty:* of direction due to the unknown state of other activities and the environment. For example, the gate-level design of a board may proceed for a while and then be disrupted by the unavailability of a chip or newly found bottlenecks in the module-level design.
- *Change:* in activities to be performed and products to be produced, requiring project flexibility and adaptability. Due to the technological nature of the engineering design activities, a large number of activities is changed along the learning curve. Often, a plan is generated in the beginning only as a guide for the future planning.

Algorithms exist which address part of the project management problem. PERT [24] and CPM [20], [23] address the scheduling problem, in particular, the detection of critical paths. Other techniques exist for the smooth assignment of resources [41]. On the other hand, few, if any, systems have addressed the problem of observing and analyzing the execution of activities, understanding how they affect other activities, and managing these effects. These are some of the issues which Callisto has addressed.

In addition to activity management, Callisto provides support to

- *product management:* maintaining a current description of the product (which is usually the outcome of a project), and determining the effects of changes to its definition (e.g., engineering change orders); and
- *resource management:* acquisition, storage, and assignment of the many resources required to support a project.

The purpose of this paper is to describe the theory of activity representation embodied in Callisto. Only a portion of this theory is described, that is, the representation of state, activity, abstraction, aggregation, time, and causality. Due to limitations in size, the representation of authority, responsibility, and possession is not included in this paper but can be found in [35].

The paper begins with an example from project management. Next, the foundation on which the theory is built is described. This foundation is a layered representation based upon the view described by Brachman [5]. Next, the two main parts of the theory are described: representation of states, activities, and goals; and the representation of time and causality. Finally, we provide a discussion of the relational abstraction.

II. A Project Management Example

Let us use an example to explain the issues involved in the semantics of project representation. Following is a typical description of a project.

> "The engineering development activity for a CPU typically involves the development of specifications, design on a CAD tool (the CAD tool is owned by the manufacturing department which uses only a portion of its capacity. The rest is used for other users and preventive maintenance. In an earlier agreement, the manufacturing department promised to give 60 percent of the CAD tool's use to the engineering department for designing Micro-84), and verification of the board on test cases. A committee of hardware engineers develops the specifications and assigns an engineer to design and verify the board specifications. Hence, specification is followed by design and verification. If verification is successful, the CPU is released for prototype development. Otherwise, the bug is located, the board is revised, and the design is performed again.
>
> "Mr. Jones, a project manager in the engineering department, has been assigned the responsibility of designing the Micro-84 CPU board. As it is not possible to cover all design aspects together, two milestones have been set for developing versions one and two of the board, respectively, and it is expected that version two of the board will conform to the project goals.
>
> "The expected duration of the design activities depends heavily on whether a new technology is used for the design or not. As the decision on whether to go with the new technology has not yet been made, two schedules need to be developed, one with the assumption that the design durations will be reduced with the help of MCA's and the other without the MCA technology."

This paragraph describes a set of activities for the de sign of the Micro-84 CPU. It describes the sequence o activities, their logical relationships, the product chang process, and the resources required. The following type of knowledge are required for scheduling or tracking th progress of these activities:

- required activities
- durations for each activity
- activity precedence
- how activities are aggregated and abstracted
- conditions under which activities can be performed, e.g., temporal relationship between specification and design (e.g., what if they overlap)
- logical connections among activities, e.g., design is done if specification is completed or if verification fails
- individualization of schedules for the two versions of the board, from a prototypical schedule
- representation of the two alternate schedules and actual dates for starting and finishing activities and for goals and milestones
- representation of changes in the product, or changes in the start or end dates (e.g., what happens when it is decided that MCA's are to be used for some portions and hence durations need to be modified to an in between level)
- resources required for each of these activities: engineers, CAD tool, simulation software, and test examples
- the period of time during which the above resources are required
- representation of constraints that restrict the usage of the resources, e.g., the maintenance schedule and previous representations by other users on the CAD machine, and the use of engineers for the next project

- interactions with the user (e.g., could he use his own terms instead of what we generate here?).

III. Layers of Representation

Given the above description, we now need to define the project concepts in terms of their attributes and relations. We need to define the engineering activities, their precedence and resource constraints, as well as aggregations, computer part descriptions, resource descriptions, ownership authority for resolving conflicts, and so on. The model should define, for each of these, their attributes, relations, and the information that flows between these concepts based on their relationships. For example, if a computer part is designed by an engineer, so are its components.

It is natural to look for commonalities among these concepts and linkages. For example, if the aggregation of activities is in any way similar to the aggregation of computer components, then a common relation can be constructed to define the common definition of aggregation, which can be specialized for the two applications. We should be able to represent the domain dependent concepts in terms of more worldly domain independent ones, e.g., the concepts of time and causality for defining precedence constraints. In this way, we can capture the underlying meaning and semantics of relations and the related flow of information. Consequently, the meaning of such models can be enhanced by combining the individual concepts to form complex concepts. We also need an implementation language for representing these concepts, their linkages, and the information flow across these relations.

The idea of a semantic representation of human knowledge originated in Quillian's thesis [29] in which concepts are represented by networks. A distinguishing feature of this work was the introduction of an "is-a" link which defines taxonomic relations and the inheritance of attributes from superconcepts to subconcepts in the hierarchy. The concept of semantic networks evolved [37], [43] and has been implemented in languages such as KLONE [4], NETL [9], and [16]. In 1975, Minsky introduced the concept of "frame." A frame partitions a semantic network into easily identifiable concepts. A variety of frame languages has been created including FRL [31], Concepts [22], KRL [3], UNITS [39], and SRL [11], [44]. A number of researchers has contributed to the semantic network approach to organizing knowledge.[2] Contributions from [5] and [12] have led to the definition of five layers of representation, as follows.

- *Domain layer* to provide concepts, words, and expressions specific to a domain of application.
- The *conceptual layer* which is comprised of models of the common primitives; such as the concepts of time, activity, state, agent, ownership, etc. These concepts are common across domains and can, therefore, be used as building blocks for modeling the domain specific concepts.
- The *epistemological layer* provides a way of regulal ing the flow of information through inheritance (describe in detail later in this section). This layer uses the concept of set, prototype, levels of aggregation, and the structura relations which link these concepts. It captures the struc tural similarities across various concepts in the conceptua layer.
- The *logical layer* defines the word *concept* as a col lection of assertions (described in detail later in this sec tion).
- The *implementation layer* which provides primitive for machine interpretation of the concepts and the asser tions.

Having provided an intuitive understanding of why eacl of these layers is needed, we will now describe these lay ers in detail. SRL [44] is the representation language usec throughout this paper. We start with the implementatioı layer and define, as we go along, building blocks used iı the subsequent layers.

A. The Implementation Layer

The purpose of the implementation layer is to define the lowest level data structures. The most basic, primitive rep resentation is a schema. Physically, a schema is composec of a schema name (printed in bold font) and a set of slots (printed in small caps). A schema is always enclosed by double braces with the schema name appearing at the top. The slots can have values assigned to them.

```
{{activity
    DURATION:
    COST:
    DESCRIPTION:}}
```

Schema 1: The Activity Schema

For example, the *activity* schema is composed of a number of slots defining attributes of the activity such as duration, cost, and description. The *Micro-84-engineering* schema defines values for each of the slots defined in the *activity* schema, e.g., cost of \$2 000 000 and duration of 2 years.

```
{{Micro-84-engineering
    creator: Mark
        INSTANCE: activity
        COST: $2,000,000
            creation-date: 1-Aug-1984
        DURATION: 2 years}}
```

Schema 2: The cpu-engineering Schema

Metainformation may be attached to any part of a schema. It provides the user with a means of documenting the information in a schema, and also for defining the semantics of schema slots and values. In the *cpu-engineering* schema, the slots in italics are metainformation attached to the schema, the slot or the value depending on their indentation. In this example, the *creator* of the schema is "Mark" and the *creation-date* of the value in the COST slot is Aug. 1, 1984.

B. The Logical Layer

The logical layer provides a logical interpretation of the information stored in the schemata. In particular, a schema–slot–value triplet is interpreted as an assertion possessed by the schema (i.e., the attribute named by the

[2] For a good review of the previous work, please refer to [5].

slot with the defined value). For example, "the project CPU-engineering costs $20 000" is an assertion. Assertions are grouped together (in a schema) to define a single concept.

C. The Epistemological Layer

The epistemological layer distinguishes types of slots and schemata. Prototype, individual, and set are distinguished schema types. Structural and taxonomic relations (e.g., *is-a*) are distinguished slots. Schemata are defined at this level with an *active* interpretation, e.g., slots and values may be inherited from one schema to another over a taxonomic relation, concepts, and their relationships.

Set, Individual, and Prototype: A **set** is a concept defined as a **collection of things that belong or are used together** [42]. An **individual** is **a member of** the **set.** The concept **set** describes the group characteristics of the individuals in the set (i.e., statistics such as number, average, etc.). A **prototype** is a concept which describes the standard or typical features of the members of a set. Thus, the concept **prototype** contains the prototypical characteristics of the individuals, while the individuals contain their individual characteristics (either exceptions to the prototypical characteristics or individual identifiers). Fig. 1 depicts the relationship among the set, the prototype, and the members of the set. The relations **member-of** and **has-member** provide an aggregation of individuals to form sets and are thus similar to the aggregation mechanisms defined later in this section. The relation **prototype-of** links a **prototype** to a **set.** The relation **is-a** and **instance** are described later in this section.

```
{{set
    IS-A: concept
    HAS-PROTOTYPE:
    HAS-MEMBER:}}
```

Schema 3: The set schema

```
{{prototype
    IS-A: concept
    PROTOTYPE-OF:}}
```

Schema 4: The prototype schema

```
{{individual
    IS-A: concept
    INSTANCE:
    MEMBER-OF:}}
```

Schema 5: The Individual Schema

```
{{prototype-of
    IS-A: relation
    INVERSE: has-prototype}}
```

Schema 6: The Prototype-of relation

Structural Relations: We would like to identify the structural relations used to structure knowledge into groups of concepts. While taxonomical links have been commonly used in the representation of knowledge since their introduction in Quillian's work [29], other ways of structuring knowledge have been explored by [5] and [12] using relations to individuate, refine, and structurally aggregate concepts. We will define these structural links and how they differ from each other. Knowledge is structured using six relations to provide **defaults, classification, elaboration, revision, individuation,** and **aggregation.**

Central to the concept of these relations is the specification of information which may be inherited from the range to the domain. Fox [11] proposed that, for two con-

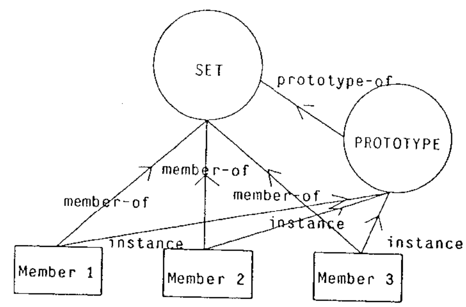


Fig. 1. The set, the prototype, and the individuals.

cepts related to each other, what is to be transferred cluded, added, and/or modified cannot be modeled w small set of classification relations (e.g., is-a, ako, vir copy). What is needed is a set of primitives which ca used to define the inheritance semantics for any relat

Brachman [6] reviewed the use of the relation **is-a** pointed to the diversity and the related confusion ir use of the relation **is-a** for semantic links (e.g., the us **is-a** for subset/superset, generalization/specification, ceptual containment, set membership, prototypes, etc.) concluded that the most prevalent use of the **is-a** rela seems to be as a **default** (assignment to a concept an default properties through the **is-a** relation). That i Clyde is an elephant, *then* he has properties typica elephants. Our approach is to identify explicitly the ferences among the various relations. Thus, the role o relation **is-a** is reduced to the definition of default p erties. Thus, if **prototype** is-a **concept,** the **assertion** the concept **prototype** inherit their default values from *relation* schema. For example, in the sentence *Jack nice guy,* the **is-a** relation is used to inherit the def mannerism for *Jack* through his association with the c cept of *nice guy*. We define the relation **is-a** to be a st tural link such that, if *A* **is-a** *B*, *A* inherits all the pro ties of *B*.[3] We define **is-a** to be reflexive (*A* **is-a** . transitive (if *A* **is-a** *B* and *B* **is-a** *C*, then *A* **is-a** *C*), . asymmetric (if *A* **is-a** *B*, *B* is not **is-a** *A*).[4] Ironically, relation **is-a** is needed to **define** itself as a relation sc to inherit all the characteristics of the concept **relati** The **instance** relation used in the transitivity for **is-a** defined later in this section. As shown below in the tr sitivity slot, *activity is-a concept* if it is possible to ge *concept* schema from *activity* schema while stepping alc at most, one **instance** relation [i.e., (repeat (step insta *t*) 0 1)] followed by none or any number of **is-a** relat steps [i.e., (repeat (step is-a *t*) 0 inf)]. The **is-a-inclusi spec** specifies that all the slots which are not listed (i. is-a, instance, is-a + inv, or instance + inv) can be herited along the is-a relation from the range to the c main of the **is-a** relation.

```
{{is-a
    IS-A: relation
    INCLUSION: is-a-inclusion-spec
    TRANSITIVITY:
      (list (repeat (step instance t) 0 1)
          (repeat (step is-a t) 0 inf))
    COMMENT: "is-a defines default"}}
```

[3]In the implementation, we had to restrict the inheritance of the inv links to avoid circular loops.

[4]Refer to [44] for the syntax of transitivity slot in **is-a** schema.

```
{{is-a-inclusion-spec
    INSTANCE: inclusion-spec
    SLOT-RESTRICTION:
      (not (or is-a instance is-a + inv instance + inv))}}
```

Schema 7: The is-a Relation

Classification is defined in Webster's dictionary [42] as *a systematic arrangement in groups or categories according to established criteria*. Classification is the process by which a set is divided or partitioned into subsets on the basis of some attribute value. It is important to note that both the domain and the range of a classification are sets. For example, *manufacturing activity* is a **subset-of** *activity* (classified on the basis of being an activity in the manufacturing domain. In the inverse process, specific sets can be combined to form more generic sets. We will use **has-subset** to relate a set (domain) to its subsets (range). The inverse of **has-subset** is **subset-of.** We will see later how this process is different in its inheritance semantics from aggregation and revision processes. In terms of inheritance semantics, **subset-of-include** shows the information that can be inherited across the subset-of relation (i.e., all slots except for subset-of and prototype-of and all the values). The relation **subset-of** is transitive, asymmetric, and nonreflexive.[5]

```
{{subset-of
    IS-A: relation
    INVERSE: has-subset
    DOMAIN: (type is-a set)
    RANGE: (schema (type is-a set))
    INCLUSION: subset-of-incl
    TRANSITIVITY: (repeat (step subset-of t) 1 inf)
    COMMENT: "subset-of defines classification"}}
```

Schema 8: The subset-of relation

```
{{subset-of-incl
    INSTANCE: inclusion-spec
    SLOT-RESTRICTION: (not (or subset-of
                      has-prototype))
    VALUE-RESTRICTION: t}}
```

Schema 9: The subset-of-inclusions

The meaning of **elaboration** as given in Webster's dictionary is *to expand something in detail*. Thus, the process of elaboration takes a concept and fills in details. Details can be appended by adding assertions (e.g., slots with values) to a concept. While classification relations operate on sets, the elaboration relation operates on individuals and prototypes. In our model, **has-elaboration** takes an individual or prototype as domain and another individual or prototype as range. The inverse of elaboration is **abstraction** which according to Webster's dictionary is *the process of reducing specific information*, and is represented by the relation **elaboration-of.** Both **elaboration-of** and **has-elaboration** are transitive, asymmetric, and reflexive.[6] The **elaboration-of-inclusion** schema defines the information that is inherited along the **elaboration-of** relation (i.e., all the slots except for elaboration-of and all the values).

[5]As defined in the transitivity slot, *manufacturing-activity is subset-of activity* if it is possible to get to the *activity* schema from the *manufacturing-activity* schema while stepping along at least one (1 to infinity) **subset-of** relation.

[6]As defined in the transitivity slot, *Micro-84-version-1 is elaboration-of Micro-84* if it is possible to get to the *Micro-84-version-1* schema from the *Micro-84* schema while stepping along zero or more (0 to infinity) **elaboration-of** relations.

```
{{has-elaboration
    IS-A: relation
    DOMAIN: (or (type is-a individual)
               (type is-a prototype))
    RANGE: (schema (or (type is-a individual)
                      (type is-a prototype)))
    INVERSE: elaboration-of
    TRANSITIVITY: (repeat (step has-elaboration t) 0 inf)}}
```

Schema 10: The Has-elaboration Relation

```
{{elaboration-of
    IS-A: relation
    DOMAIN: (or (type is-a individual)
               (type is-a prototype))
    RANGE: (schema (or (type is-a individual)
                      (type is-a prototype)))
    INVERSE: has-elaboration
    INCLUSION: elaboration-of-inclusion
    TRANSITIVITY: (repeat (step elaboration-of t) 0 inf)
    COMMENT: "elaboration-of defines abstraction"}}
```

Schema 11: The Elaboration-of Relation

```
{{elaboration-of-inclusion
    IS-A: relation
    SLOT-RESTRICTION: (not elaboration-of)
    VALUE-RESTRICTION: t}}
```

Schema 12: The Elaboration-of-inclusion

Aggregation is *to collect or gather into a whole.* The emphasis in aggregation is toward combining the parts to make a whole. The parts could belong to different sets or instances of sets. The disaggregates are **part-of** the aggregate concept. Parts inherit some attributes from their aggregation (e.g., ownership), others are aggregated (e.g., cost), or averaged (e.g., performance). For example, *CPU-specification* is **part-of** *the CPU-engineering-network*. The inverse of **part-of** is **has-part**. The **part-of** relation is reflexive as well as transitive, although asymmetric (similar to the **elaboration-of** relation, described above).

```
{{part-of
    IS-A: relation
    DOMAIN: (or (type is-a individual)
               (type is-a prototype))
    RANGE: (schema (or (type is-a individual)
                      (type is-a prototype)))
    INVERSE: has-part
    TRANSITIVITY: (repeat (step part-of t) 0 inf)
    COMMENT: "part-of defines aggregation"}}
```

Schema 13: The Part-of Relation

Revision as defined in Webster's dictionary is *to make a new amended, improved, or up-to-date version*. Thus, the process of revision converts a range object into a domain object by adding improvements in its representation. Here, **both the range and the domain need to be at the same level of aggregation and belong to the same set of concepts** for a meaningful revision. Revisions can be introduced by adding or transforming slots. For example, *version 2 of Micro-84* is a **revision-of** *version 1*. Both version 1 and 2 are at the same level of aggregation. As opposed to elaboration, **revision** is a transformation process, and thus describes a progression in time. The inverse link is **revised-by** and it does not conceptually represent a process. The relation **revision-of** is transitive, asymmetric, and nonreflexive (similar to the **subset-of** relation).

```
{{revision-of
    IS-A: relation
    DOMAIN: (or (type is-a prototype)
               (type is-a individual))
    RANGE: (schema (or (type is-a prototype)
                      (type is-a individual)))
    INVERSE: revised-by
    TRANSITIVITY:
     (repeat (step revision-of t) 1 inf)}}
```

Schema 14: The Revision-of Relation

Individuation is *the development of the individual from the universal* [5] and is represented by the **instance** rela-slot with the defined value). For example, "the project CPU-engineering costs $20 000" is an assertion. Assertions are grouped together (in a schema) to define a single concept.

C. The Epistemological Layer

The epistemological layer distinguishes types of slots and schemata. Prototype, individual, and set are distinguished schema types. Structural and taxonomic relations (e.g., *is-a*) are distinguished slots. Schemata are defined at this level with an *active* interpretation, e.g., slots and values may be inherited from one schema to another over a taxonomic relation, concepts, and their relationships.

Set, Individual, and Prototype: A **set** is a concept defined as a **collection of things that belong or are used together** [42]. An **individual** is **a member of** the **set.** The concept **set** describes the group characteristics of the individuals in the set (i.e., statistics such as number, average, etc.). A **prototype** is a concept which describes the standard or typical features of the members of a set. Thus, the concept **prototype** contains the prototypical characteristics of the individuals, while the individuals contain their individual characteristics (either exceptions to the prototypical characteristics or individual identifiers). Fig. 1 depicts the relationship among the set, the prototype, and the members of the set. The relations **member-of** and **has-member** provide an aggregation of individuals to form sets and are thus similar to the aggregation mechanisms defined later in this section. The relation **prototype-of** links a **prototype** to a **set.** The relation **is-a** and **instance** are described later in this section.

```
{{set
   IS-A: concept
   HAS-PROTOTYPE:
   HAS-MEMBER:}}
```

Schema 3: The set schema

```
{{prototype
   IS-A: concept
   PROTOTYPE-OF:}}
```

Schema 4: The prototype schema

```
{{individual
   IS-A: concept
   INSTANCE:
   MEMBER-OF:}}
```

Schema 5: The Individual Schema

```
{{prototype-of
   IS-A: relation
   INVERSE: has-prototype}}
```

Schema 6: The Prototype-of relation

Structural Relations: We would like to identify the structural relations used to structure knowledge into groups of concepts. While taxonomical links have been commonly used in the representation of knowledge since their introduction in Quillian's work [29], other ways of structuring knowledge have been explored by [5] and [12] using relations to individuate, refine, and structurally aggregate concepts. We will define these structural links and how they differ from each other. Knowledge is structured using six relations to provide **defaults, classification, elaboration, revision, individuation,** and **aggregation.**

Central to the concept of these relations is the specification of information which may be inherited from the range to the domain. Fox [11] proposed that, for two con-

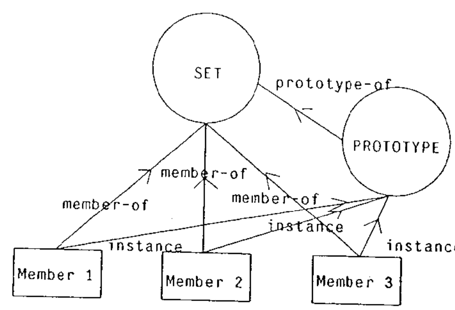


Fig. 1. The set, the prototype, and the individuals.

cepts related to each other, what is to be transferre cluded, added, and/or modified cannot be modeled small set of classification relations (e.g., is-a, ako, v copy). What is needed is a set of primitives which used to define the inheritance semantics for any rel

Brachman [6] reviewed the use of the relation **is-** pointed to the diversity and the related confusion use of the relation **is-a** for semantic links (e.g., the **is-a** for subset/superset, generalization/specification ceptual containment, set membership, prototypes, etc concluded that the most prevalent use of the **is-a** re seems to be as a **default** (assignment to a concept a default properties through the **is-a** relation). That Clyde is an elephant, *then* he has properties typi elephants. Our approach is to identify explicitly th ferences among the various relations. Thus, the role relation **is-a** is reduced to the definition of default erties. Thus, if **prototype** is-a **concept,** the **assertic** the concept **prototype** inherit their default values fro *relation* schema. For example, in the sentence *Jack nice guy,* the **is-a** relation is used to inherit the d mannerism for *Jack* through his association with the cept of *nice guy*. We define the relation **is-a** to be a s tural link such that, if A **is-a** B, A inherits all the pr ties of B.[3] We define **is-a** to be reflexive (A **is-a** transitive (if A **is-a** B and B **is-a** C, then A **is-a** C), asymmetric (if A **is-a** B, B is not **is-a** A).[4] Ironicall relation **is-a** is needed to **define** itself as a relation to inherit all the characteristics of the concept **rela** The **instance** relation used in the transitivity for **is** defined later in this section. As shown below in the t sivity slot, *activity is-a concept* if it is possible to g *concept* schema from *activity* schema while stepping a at most, one **instance** relation [i.e., (repeat (step inst *t*) 0 1)] followed by none or any number of **is-a** rel steps [i.e., (repeat (step is-a *t*) 0 inf)]. The **is-a-inclus spec** specifies that all the slots which are not listed (is-a, instance, is-a + inv, or instance + inv) can b herited along the is-a relation from the range to the main of the **is-a** relation.

```
{{is-a
   IS-A: relation
   INCLUSION: is-a-inclusion-spec
   TRANSITIVITY:
     (list (repeat (step instance t) 0 1)
        (repeat (step is-a t) 0 inf))
   COMMENT: "is-a defines default"}}
```

[3]In the implementation, we had to restrict the inheritance of the ir links to avoid circular loops.

[4]Refer to [44] for the syntax of transitivity slot in **is-a** schema.

```
{{is-a-inclusion-spec
    INSTANCE: inclusion-spec
    SLOT-RESTRICTION:
      (not (or is-a instance is-a + inv instance + inv))}}
```

Schema 7: The is-a Relation

Classification is defined in Webster's dictionary [42] as *a systematic arrangement in groups or categories according to established criteria*. Classification is the process by which a set is divided or partitioned into subsets on the basis of some attribute value. It is important to note that both the domain and the range of a classification are sets. For example, *manufacturing activity* is a **subset-of** *activity* (classified on the basis of being an activity in the manufacturing domain. In the inverse process, specific sets can be combined to form more generic sets. We will use **has-subset** to relate a set (domain) to its subsets (range). The inverse of **has-subset** is **subset-of.** We will see later how this process is different in its inheritance semantics from aggregation and revision processes. In terms of inheritance semantics, **subset-of-include** shows the information that can be inherited across the subset-of relation (i.e., all slots except for subset-of and prototype-of and all the values). The relation **subset-of** is transitive, asymmetric, and nonreflexive.[5]

```
{{subset-of
    IS-A: relation
    INVERSE: has-subset
    DOMAIN: (type is-a set)
    RANGE: (schema (type is-a set))
    INCLUSION: subset-of-incl
    TRANSITIVITY: (repeat (step subset-of t) 1 inf)
    COMMENT: "subset-of defines classification"}}
```

Schema 8: The subset-of relation

```
{{subset-of-incl
    INSTANCE: inclusion-spec
    SLOT-RESTRICTION: (not (or subset-of
                        has-prototype))
    VALUE-RESTRICTION: t}}
```

Schema 9: The subset-of-inclusions

The meaning of **elaboration** as given in Webster's dictionary is *to expand something in detail*. Thus, the process of elaboration takes a concept and fills in details. Details can be appended by adding assertions (e.g., slots with values) to a concept. While classification relations operate on sets, the elaboration relation operates on individuals and prototypes. In our model, **has-elaboration** takes an individual or prototype as domain and another individual or prototype as range. The inverse of elaboration is **abstraction** which according to Webster's dictionary is *the process of reducing specific information*, and is represented by the relation **elaboration-of.** Both **elaboration-of** and **has-elaboration** are transitive, asymmetric, and reflexive.[6] The **elaboration-of-inclusion** schema defines the information that is inherited along the **elaboration-of** relation (i.e., all the slots except for elaboration-of and all the values).

[5]As defined in the transitivity slot, *manufacturing-activity is subset-of activity* if it is possible to get to the *activity* schema from the *manufacturing-activity* schema while stepping along at least one (1 to infinity) **subset-of** relation.

[6]As defined in the transitivity slot, *Micro-84-version-1 is elaboration-of Micro-84* if it is possible to get to the *Micro-84-version-1* schema from the *Micro-84* schema while stepping along zero or more (0 to infinity) **elaboration-of** relations.

```
{{has-elaboration
    IS-A: relation
    DOMAIN: (or (type is-a individual)
              (type is-a prototype))
    RANGE: (schema (or (type is-a individual)
                     (type is-a prototype)))
    INVERSE: elaboration-of
    TRANSITIVITY: (repeat (step has-elaboration t) 0 inf)}}
```

Schema 10: The Has-elaboration Relation

```
{{elaboration-of
    IS-A: relation
    DOMAIN: (or (type is-a individual)
              (type is-a prototype))
    RANGE: (schema (or (type is-a individual)
                     (type is-a prototype)))
    INVERSE: has-elaboration
    INCLUSION: elaboration-of-inclusion
    TRANSITIVITY: (repeat (step elaboration-of t) 0 inf)
    COMMENT: "elaboration-of defines abstraction"}}
```

Schema 11: The Elaboration-of Relation

```
{{elaboration-of-inclusion
    IS-A: relation
    SLOT-RESTRICTION: (not elaboration-of)
    VALUE-RESTRICTION: t}}
```

Schema 12: The Elaboration-of-inclusion

Aggregation is *to collect or gather into a whole.* The emphasis in aggregation is toward combining the parts to make a whole. The parts could belong to different sets or instances of sets. The disaggregates are **part-of** the aggregate concept. Parts inherit some attributes from their aggregation (e.g., ownership), others are aggregated (e.g., cost), or averaged (e.g., performance). For example, *CPU-specification* is **part-of** *the CPU-engineering-network.* The inverse of **part-of** is **has-part**. The **part-of** relation is reflexive as well as transitive, although asymmetric (similar to the **elaboration-of** relation, described above).

```
{{part-of
    IS-A: relation
    DOMAIN: (or (type is-a individual)
              (type is-a prototype))
    RANGE: (schema (or (type is-a individual)
                     (type is-a prototype)))
    INVERSE: has-part
    TRANSITIVITY: (repeat (step part-of t) 0 inf)
    COMMENT: "part-of defines aggregation"}}
```

Schema 13: The Part-of Relation

Revision as defined in Webster's dictionary is *to make a new amended, improved, or up-to-date version*. Thus, the process of revision converts a range object into a domain object by adding improvements in its representation. Here, **both the range and the domain need to be at the same level of aggregation and belong to the same set of concepts** for a meaningful revision. Revisions can be introduced by adding or transforming slots. For example, *version 2 of Micro-84* is a **revision-of** *version 1*. Both version 1 and 2 are at the same level of aggregation. As opposed to elaboration, **revision** is a transformation process, and thus describes a progression in time. The inverse link is **revised-by** and it does not conceptually represent a process. The relation **revision-of** is transitive, asymmetric, and nonreflexive (similar to the **subset-of** relation).

```
{{revision-of
    IS-A: relation
    DOMAIN: (or (type is-a prototype)
              (type is-a individual))
    RANGE: (schema (or (type is-a prototype)
                     (type is-a individual)))
    INVERSE: revised-by
    TRANSITIVITY:
     (repeat (step revision-of t) 1 inf)}}
```

Schema 14: The Revision-of Relation

Individuation is *the development of the individual from the universal* [5] and is represented by the **instance** relation. It can be interpreted as a copy of the prototype with an individual name and exceptions, if any. For example, *CPU-engineering* is the process of engineering development of a CPU, while *CPU-engineering%1 is an* **instance** of *CPU-engineering for building the first version of Micro-84 CPU.*

```
{{instance
    IS-A: relation
    DOMAIN: (type is-a individual)
    RANGE: (schema (type is-a prototype))
    INCLUSION: instance-inclusion}}
```

Schema 15: The Instance Schema

```
{{instance-inclusion
    INSTANCE: inclusion-spec
    SLOT-RESTRICTION: (not (or prototype-of subset-of is-a
                       is-a + inv instance + inv))
    VALUE-RESTRICTION: t}}
```

Schema 16: The Instance-inclusion spec

As we go on to develop relations for specialized needs, we find that these relations can inherit the inheritance semantics from more generic relations. For example, if the aggregation process in objects is similar to the aggregation process in activities, then their commonalities can be represented using a domain independent **part-of** relation, from which each of the relations, specific to activities and objects, inherits the common inheritance semantics and adds to it what is specific to activities or objects. Thus, we begin to build a hierarchy of these relations, starting from the most general concepts like classification and abstraction, to more and more specific relations. Such relations (e.g., **sub-activity-of** and **sub-state-of,** in Section IV-A and IV-B, respectively) are defined in the semantic layer.

D. *The Semantic Layer*

The semantic layer contributes to the depth of representation by facilitating inheritance of the underlying common knowledge. For example, all types of activities, whether design or verification, engineering or manufacturing, share common information, such as cost, duration, and responsibility. They have similar underlying notions of causality, time relationships, resource possessions, and milestones. We therefore need a common definition of activity which can be used for further defining specific activities.

The concepts in the semantic layer can be classified into three major categories: action related, object related, and agent related. The action related primitives include concepts of activity, state, causation, and temporal relations. The definition of object includes its refinements and disaggregations and the theory of change. Constraints can be imposed on the definition of action or object related primitives. The agents possess and own objects and are organized through authority structures.

In the following sections, we will describe, in detail, the definition and representation for activity, state, time, and causality. We will build a theory for each of these concepts which brings forth a general definition of the concept. The semantic layer is defined using the concepts of inheritance and structure defined in the epistemological layer.

E. *The Domain Layer*

For the project management example (see Sectio we need to define the concepts of specification, desig verification activities, and their relationships, the puter parts, the engineering and manufacturing d ments, and the contracts between them on the usage CAD machine. These terms can now be defined more easily using the epistemological concepts and t mantic definition of activities, objects, and agents addition of a new domain only requires the additi domain specific concepts and their definition in ter the epistemological and semantic layers.

IV. The Theory of Activity, State and Goa

Much of Callisto's capabilities rely upon de knowledge of both activities and the conditions which they can be performed. For example, plannin quires a representation for each activity, and know of resources consumed and produced by each activi order to select and deduce precedence (i.e., seq them). To support hierarchical reasoning, activities be represented at multiple levels of abstraction. Sch ing uses the same knowledge as planning, but in adc requires time information, and knowledge of alterna (e.g., activities, substitutable resources) for situatio which certain resources are not available at the spec time. Chronicling is the facility for specifying activity tus. It analyzes the implementation of the schedules tects problems, such as deviations and interactions, attempts to repair them. In order to perform this task chronicling system must distinguish among various sions of activities, including the predicted ones create scheduling and the actual ones performed by the pro It must also have knowledge of how the predicted ac ties constrain the project and what must be done to re any deviations.

A. *Theory of Activity*

First, we need to define the concept of activity. definition should include the type of tasks that ca called activities, relationships among them and with p ect goals, and issues of aggregation and abstraction the example

> . . . a project manager has been assigned the respo sibility of designing the Micro-84 CPU Board. Thi design involves development of specifications, de sign on a CAD tool, and verification of the board o test cases.

Are all of these activities? How is the overall project lated to these activities? How are the goals set? Fina how is their disaggregation done by the project mana and by others in the organization?

Considerable research has been done in defining the lating activities or acts in natural language systems, pr lem solving systems, and in linguistics and philosop

[7]For an excellent review, please refer to [2].

These works provide useful insights into the hierarchical representation of activities and in representing the prerequisites and results of an activity. Allen [2] has developed a theory of action, which is by far the most general and includes actions involving nonactivity, actions which are not easily decomposable, and actions which occur simultaneously.

We define the **activity** as the basic unit of action in the project management environment. The project manager starts with a **project activity**[8] assigned to him, disaggregates the project into a set of subactivities, the execution or completion of which leads to the completion of the project. An **activity** is a transformation of the world from one situation or state to another [25] which, directly or indirectly, carries the project from the starting state toward the goal state.

1) Aggregation and Abstraction: Activities are often defined at many levels of abstraction. Sacerdoti [32], [33] constructed a system which stratified activities by the removal of conditions. The choice of condition was based on "importance." In the NONLIN system, Tate [40] developed a "task formalism," which described various actions, preconditions, and precedences. The NONLIN system expanded these high-level descriptions into detailed plans. In the task formalism, the supervised conditions were differentiated from others as they involved details that could be expanded by the planning system (and thus, involved no interaction with the other high-level activities). In order to facilitate different levels of aggregation, [13] used **sub-activity-of** which provides disaggregation of activities. The relation **refined-by** was used by some researchers [8], [12] to connect activities to their detailed counterparts.

We use the epistemological layer concepts to model the relationship between *CPU-engineering* and *CPU-specification*, *CPU-design*, and *CPU-verification* activities in our example. None of the relations, mentioned by the researchers above, seems appropriate for relating *CPU-design* to *CPU-specification*. It is not just an elaboration, because elaboration involves an expansion of an object into another, where both are at the same level of aggregation. It is not a disaggregation (as implicitly stated in the **sub-activity-of** relation of Goldstein), because *CPU-engineering* is not at the same level of detail (or abstraction) as *CPU-specification*. In other words, the different levels of specificity [32], [40] and and/or aggregation [17] coexist in the specification of activities.

Thus, we should be using both aggregation and elaboration. An activity is **elaborated** to an **aggregate activity** (an activity network), which then has activities, which are **part-of** the aggregate activity. For example, the *CPU-engineering* activity has an elaboration, *CPU-engineering-network* which, in turn, has three activities *CPU-specification*, *CPU-design*, and *CPU-verification* as **part-of** *CPU-engineering-network*. The **elaboration-of** relation helps in the separation of the single activity, *CPU-engineering*, from its detailed description, thus facilitating descriptions at different levels of abstraction or multiple elaborations of the same activity. For example, as in [40], the activity *CPU-engineering* describes all the interactions with the other activities (outside the *CPU-engineering-network*), while the interactions within the *CPU-engineering-network* are hidden at the level of the high-level activity, *CPU-engineering* (see Section IV-B and Fig. 2).

We will discuss inheritance issues related to activity aggregation next, and issues related to temporal aggregation in Section V-A. The SRL representation of the concept **activity** is as follows:

```
{{activity
    ELABORATION-OF:
        Range: (schema (type is-a activity))
    HAS-ELABORATION:
        Range: (schema (type is-a activity))
    PART-OF:
        Range: (schema (type is-a activity))
    COST:
    DURATION:}}
```

Schema 17: The Activity Schema

An activity should inherit information from other activites in higher and lower levels of abstraction. For example, if the activity *CPU-engineering* is the responsibility of a project manager, he is also responsible for *CPU-specification*, *CPU-design*, and *CPU-verification* activities. Also, the cost of executing *CPU-engineering* should be the aggregation of the cost of its lower level activities. As these various types of inheritances are specific to the activity world, it is inappropriate to include them in the definition of the **part-of** relation. We define the **sub-activity-of** relation, which acts like **part-of,** for aggregating activities. Its inverse is the relation **has-sub-activity.**

There are two types of information flow across the aggregation levels. First, the inheritance of information by lower level activities from the higher levels. Inheritance flows from the range to the domain via the **sub-activity-of** relation and the inclusion specifications in SRL. Second, the higher level activities aggregate information (e.g., cost) from lower levels (through a many-to-one map specification, has-sub-activity-map).[9] This aggregation of information can be represented in the **sub-activity-of** relation:

```
{{sub-activity-of
    IS-A: part-of
    INVERSE: has-sub-activity
    INCLUSION: sub-activity-of-incl
    DOMAIN:
        Range: (type is-a activity)
    RANGE:
        Range: (type is-a activity)}}
```

Schema 18: The sub-activity-of Relation

```
{{sub-activity-of-incl
    IS-A inclusion-spec
    SLOT-RESTRICTION:(or priority responsibility-of)}}
```

Schema 19: Inclusions in sub-activity-of

```
{{has-sub-activity
    IS-A: has-part
    MAP: has-sub-activity-map}}
```

Schema 20: The has-sub-activity Relation

[8]A project activity in the engineering design context starts with a plan to produce a new product and ends with the first revenue shipment of the product. It has a goal to design the product; while the starting point is an abstract concept in the mind of the design initiator.

[9]We will later return to aggregation (of activity status) while describing the representation of state.

Here, **has-sub-activity-map** defines what can be aggregated along the **has-sub-activity** relation, while **sub-activity-of-incl** defines the information that can be inherited along the **sub-activity-of** relation. The schema description of **sub-activity-of-incl** states that the slots "priority" and "responsibility-of" can be inherited by a subactivity from its superactivity.

The **has-elaboration** relations can be used to link an abstract activity to a detailed activity network. These relations are useful in multiuser communication situations where an activity at one level of description needs to be elaborated into its components at a lower level. For example, the engineering manager thinks of *CPU-engineering* as a single activity with no further disaggregations. The same activity is an aggregate activity further decomposed into *CPU-specification*, *CPU-design*, and *CPU-verification* activities in the eyes of the project manager dealing with these activities. The relation **elaboration-of,** which relates a detailed aggregate activity (*CPU-engineering-network*) to the abstract activity (*CPU-engineering*), suffices in its inheritance definition, as it inherits all information from the abstract to the elaborated concept (see Fig. 2).

How are the activities aggregated? It is not necessary for the aggregation to the conjunctive only. In real life, very often managers refer to disjunct aggregations. (*The design can be done either by design on a CAD machine or on a bread-board.*) The **aggregate activity,** therefore, could be a conjunctive or disjunctive aggregation of its components. The schema for aggregate activity contains a type slot to provide this information.

```
{{aggregate-activity
    IS-A: activity
    TYPE:
        Range: (or "and" "or" "xor")
    HAS-SUB-ACTIVITY:
    ELABORATION-OF:}}
```

Schema 21: The Aggregate Activity Schema

```
{{cpu-engineering-network
    IS-A: aggregate-activity
    TYPE: "and"
    HAS-SUB-ACTIVITY: cpu-specification cpu-design
            cpu-verification
    ELABORATION-OF: cpu-engineering}}
```

Schema 22: The cpu-engineering-network Schema

Is it necessary for an activity to be part of one and only one aggregate activity? While the project manager considers *CPU-specification*, *CPU-design*, and *CPU-verification* as parts of a project, a design engineer would probably consider the *CPU-design* as a part of various design activities to be done. In the organizational environment, it is common to find that quality control and the material departments aggregate activities in different ways. Similarly, there can be multiple elaborations of the same activity, each emphasizing different aspects of the activity. For example, the overall activity *CPU-engineering* may have altogether different components for the CAD and physical space designers, respectively. Each of these elaborations refers to the same abstract activity. As specified, the activity representation is capable of dealing with multiple ways of aggregation and elaboration.

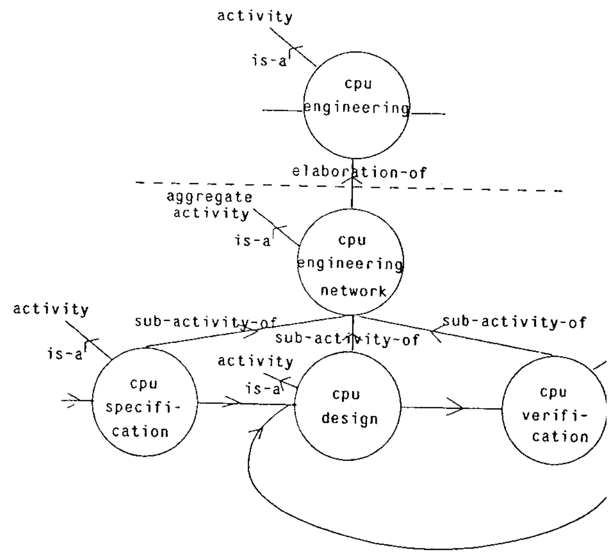


Fig. 2. Activity aggregation and abstraction.

B. Theory of State

The next problem to be settled is the representa conditions under which an activity may be performe the new conditions produced by the activity. In c ample, *CPU-design* activity is started when *CPU-s cation* is completed or if *CPU-verification* fails. project management tasks, such arbitrarily comple ditions involving logical constructs should be repre by the activity model.

Let us start with the definition of a state. Hendri described a state of the world model "like a still graph of a dynamic situation, representing objects a relationships among objects as they exist the mome photograph is taken." In project management, we that the concept of state (or event as used in PERT models [20], [23]) was even more general and inc state of beings over time (similar to the definition uations in [17]). Thus, state defines a fact which hc of some point in time (e.g., *CPU-specification* is com or for a period of time (e.g., possession of CAD ma for the duration of *CPU-design*).[10]

1) State Aggregation: In the world of project a ties, we would like to use states as a way of represe alternative scenarios or situations in which an activit be executed, as well as the resulting alternative outcc Thus, using superimposed logical structures [17], difi scenarios required for executing the activity are com to form a composite state, which **enables** the activity overall logical structure *holds* whenever any of its stituent alternative situations *holds.* We therefore h relative representation of the type "if . . . then sta activity." For example, *CPU-design* can be done

[10]We would like to point out here that PERT/CPM representati nored the state of being over time in their representation of events. ever, the only difference between the two is temporal. As we have sociated temporal issues from the causal issues, it is now possible to cc them and thus use a more general view of state. We will discuss the underlying temporal differences later in Section V-A.

completing CPU-specification, or if *CPU-verification fails* and *requires a CAD machine*. This implies that *CPU-design* can not be executed unless this composite state of the world is **true.**[11]

The new condition produced by the activity is the **caused state** which is an aggregation of different alternatives caused by the activity. The schema representation of the activity can now be extended to include the state links

```
{{activity
    HAS-SUB-ACTIVITY:
        Range: (schema (type is-a activity))
    SUB-ACTIVITY-OF:
        Range: (schema (type is-a activity))
    ENABLED-BY:
        Range: (schema (type is-a state))
    CAUSE:
        Range: (schema (type is-a state))
    COST:
    DURATION: }}
```

Schema 23: The Modified Activity Schema

The relations **enabled-by** and **cause** (which link an activity to its enabling and caused states, respectively) are defined later in Section V-B.

Let us now look at the *CPU-design* activity in the example introduced earlier. The schema representation of the *CPU-design* activity is given below. The aggregated enabling state is *start-CPU-design* which enables the *CPU-design activity*. The *CPU-design-complete* state is caused by the *CPU-design* activity and represents the logical aggregation of the possible alternative outcomes.

```
{{cpu-design
    IS-A: activity
    ENABLED-BY: start-cpu-design
    CAUSE: cpu-design-complete
    SUB-ACTIVITY-OF: cpu-engineering-network
    COST: 200,000
    DURATION: 120 days}}
```

Schema 24: The cpu-design Activity Schema

Let us look at the example again.

> The design (is done) on a CAD tool . . . specification is followed by design . . . if verification fails . . . design is performed again.

Thus, *CPU-design* is done when *CPU-specification* is completed or *CPU-verification* fails *and requires a CAD machine* for the duration of the *CPU-design*. We need to disaggregate *start-CPU-design* to represent these logical relationships. As with the aggregation of the activities, the aggregation of states can also be accomplished by the **part-of** relation or its elaboration. We use the **has-sub-state** relation (with its inverse **sub-state-of**) to link an aggregate state to its disaggregates. Hence, possession of the CAD machine is a substate of the enabling state *start-CPU-design*. The relation **has-sub-state** can be used to determine whether the composite state **holds** (this is done by associating the logic of state propagation with the map-spec of the **has-sub-state** relation in SRL). The **sub-state-of** relation is a **part-of** with the addition of the appropriate truth propagation algorithm (described later in Section V-B).

[11]The problem of causality is dealt with in Section V-B, which gives a definition of the "true" state, its propagation as well as the roles of relations **enabled-by** and **cause.**

```
{{sub-state-of
    IS-A: part-of
    INVERSE: has-sub-state
    DOMAIN:
        Range: (schema (type is-a state))
    RANGE:
        Range: (schema (type is-a state))
    INTRODUCTION: sub-state-propagation-action}}
```

Schema 25: The sub-state-of relation

States of the world represent completion of activites, possession of resources, milestones that must be met, their aggregations, etc. While any of these states could be associated with an activity, their roles and characteristics differ. For example, the possession of resources is represented by states which hold (or are "true") for a duration of time, while completion of activities is a one-shot situation [30]. A **classification of states** is required to properly represent the different types of logical preconditions and aggregations. Fig. 3 depicts this classification which shows two major classes of states—**aggregate** states and **leaf** states. The aggregate state could be an **or** (disjunct), which is **true** if any of its substates is **true,** or an **and** (conjunct), where all of its substates should be **true to** make the **and** *state* **true.** The leaf states are further classified into **status predicates,** depicting facts related to activities status, and **possess predicates,** depicting possession of resources for the duration of the activity. The rationale for differentiating between status predicates and possess predicates will be discussed later in the theory of time (see Section V-A).

Returning to our example, the states—*CPU-spec-complete* and *CPU-verification-failed*—are aggregated by a disjunct *or-CPU-design state*. The state *start-CPU-design* is a conjunct of the *or-CPU-design* and *possess-CAD-machine*. The schema representation of these states is as follows:

```
{{start-cpu-design
    IS-A: and-state
    ENABLE: cpu-design
    HAS-SUB-STATE: or-cpu-design possess-CAD-machine
```

Schema 26: The start-cpu-design Schema

```
{{or-cpu-design
    IS-A: or-state
    SUB-STATE-OF: start-cpu-design
    HAS-SUB-STATE: cpu-spec-complete
                   cpu-verification-failed}}
```

Schema 27: The or-cpu-design Schema

```
{{possess-CAD-machine
    IS-A: possess-predicate
    SUB-STATE-OF: start-cpu-design
    REQUIRE: CAD-machine
    RESOURCE-UTILIZATION: 100}}
```

Schema 28: The possess-CAD-machine Schema

We refer to this aggregation of states as *state trees*. Fig. 4 shows the enabling state tree described before. The enabling state tree, the activity and the caused state tree together define when an activity can be done, what it does and the results it delivers. An **activity cluster** is an aggregate concept composed of an activity, an enabling tree, and a caused tree. Later sections will further describe this concept and its use as a partition [17].

2) State Abstraction: There is a need to map state information across the levels of activity hierarchy, thus easing the process of project monitoring. For example, the

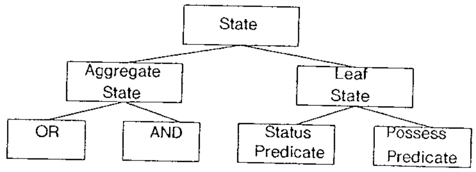


Fig. 3. The state classification hierarchy.

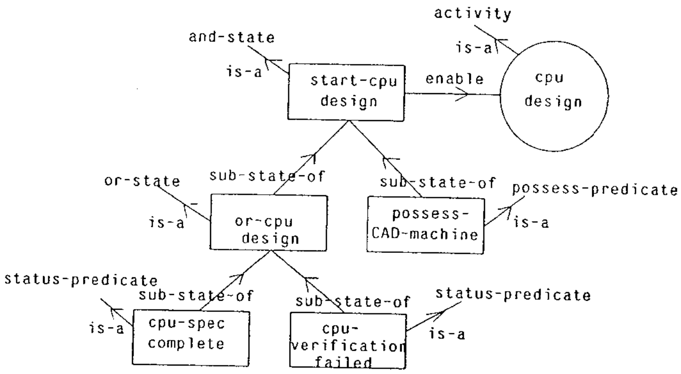


Fig. 4. State aggregation for CPU design.

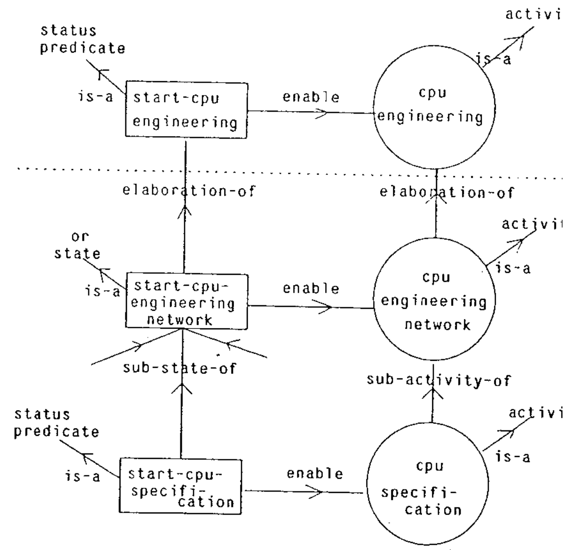


Fig. 5. Aggregation and abstraction of states.

abstract activity, *CPU-engineering*, starts when *CPU-specification* is started, and is completed when *CPU-verification* is completed successfully.

The abstraction of state information is almost identical to the abstraction of activities described before in Section IV-A. We need to map the starting of the disaggregate activities to the starting of the abstract activity. Thus, the enabling states of the initial activities form an enabling network, using **sub-state-of** with conjuncts and disjuncts. This enabling network is an elaboration of the enabling state of the abstract activity. For example, if *CPU-specification* were a possible starting point for the *CPU-engineering activity*, the *start-CPU-engineering-network*, maps the start of *start-CPU-engineering* to the start of *CPU-specification* (see Fig. 5).

```
{{or-state
    IS-A: aggregate-state
    HAS-SUB-STATE:
    SUB-STATE-OF:
    ELABORATION-OF:}}
```

Schema 29: The Or State Schema

```
{{start-cpu-engineering-network
    IS-A: or-state
    HAS-SUB-STATE: start-cpu-specification
                   start-cpu-design
    ELABORATION-OF: start-cpu-engineering}}
```

Schema 30: The start-design-CPU-network Schema

C. Goals

The project management task begins with a statement of goals. These goals guide the construction of the project's activity/state network. Two basic types of goals have been distinguished: goals which define the milestone states on the performance of the project and its completion, and goals which constrain performance of activities (e.g., constraints on time or money to spend on an activity).

State goals are represented in the same form as states. The top of this hierarchy defines the project goals, breaking them into milestones for smaller time periods structure of the goal hierarchy is similar to the stru of the state hierarchy, with **part-of** relations to pr aggregations and **elaboration-of** to elaborate the goa a network of goals/milestones. A network of goals gregated into an **aggregate-goal** using the **part-of** tion, wherein goal information can be summed or aged.

```
{{goal-state
    IS-A: state}}
```

Schema 31: The Goal-state schema

```
{{aggregate-goal
    TYPE:
        Range: (or and or xor)
    HAS-PART:
        Range: (type is-a goal-state)
    ELABORATION-OF:
        Range: (type is-a goal-state)}}
```

Schema 32: The Aggregate Goal Schema

These goal states also need to be linked to the activ Whenever an activity is linked to a goal state, its con tion must lead to the satisfaction of the goal state. example, in Fig. 6, *Milestone-2* is a **goal-state** linke the activity, *CPU-engineering*. Whenever, *CPU-engin ing* is completed, it should satisfy the specificatior *Milestone-2*. The relation **must-satisfy** is used to li caused state to a milestone state.

```
{{must-satisfy
    IS-A: relation
    DOMAIN: (type is-a state)
    RANGE: (type is-a goal-state)}}
```

Schema 33: The Must-satisfy Relation

Activity goals such as the cost, the end-time (to be plained in Section V-C), and the resources produce an activity, are specified as bounds on these values (e minimum and maximum admissible values). These g act as constraints and are attached (in the form metaschema) to the affected slot. Thus, the cost goa attached to the cost slot:

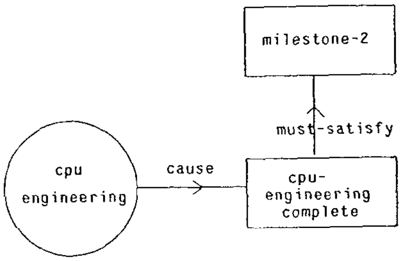


Fig. 6. The activity goals.

```
{{activity
   COST:
      Constrained-by: (type "is-a" "goal-constraint")}}
```

Schema 34: The Activity Schema with Cost goal

```
{{goal-constraint
   IS-A: constraint
   IMPORTANCE:
   VALUE:}}
```

Schema 35: The Goal Constraint

Section IV-D describes how these goals are used in conjunction with schedules to monitor activities. The details of constraint specification and usage can be found in [12].

D. *Instantiation and Manifestation*

The next step toward the construction of a theory of activity, states, and goals is providing the representational capability to differentiate between prototypical networks, individualized networks, schedules and actual completion reports. In our interviews with managers, we found that they had the notion of a prototypical network which they used repeatedly for similar design tasks. For each task, they used the prototypical network possibly with some task specific variations (e.g., *everything but the power supply design activities*). A schedule was generated before starting a task and updated at the end of each milestone (referred to as *schedule of Jan. 15*, *schedule of June 30*, ...). Finally, they create activity completion reports providing the actual start and completion dates for the activities. The project managers relate and enquire about relative location of activities (e.g., *what do I do after design*), about a schedule (e.g., *when does design start in the new schedule*), across schedules (e.g., *how much will the slip be now, compared to the last schedule*), or comparing schedules with actual progress (e.g., *how much did we slip in completing design of CPU*). Needless to say, there is more than one representation of an activity and a need for linking these diverse representations.

Organizations typically maintain standard procedures (e.g., *Engineering Guidelines*) which describe the procedure or the activities involved in a task. Even when standard procedures are not maintained formally, people have rich sets of past experiences or scripts [36] stored as prototypical activities and states. For a new task or project, these standard procedures or past experiences are individuated and the new task becomes an instance of the standard procedures. In effect, the set of activities which comprise the new task is linked by an instance relation to the corresponding activities and states in the standard procedures.[12] For example, *CPU-design%1* is generated as an instance of *CPU-design*, enabled by *start-CPU-design%1* and causes *CPU-design-complete%1* where *start-CPU-design%1* and *CPU-design-complete%1* are instances of *start-CPU-design* and *CPU-design-complete*, respectively.

```
{{cpu-design%1
   INSTANCE: cpu-design
   ENABLED BY: start-cpu-design%1
   CAUSE: cpu-design-complete%1}}
```

Schema 36: The cpu-design%1 schema

Each activity in the individuated activity network is an instance of a prototypical activity. In other words, the activities are defined elsewhere, and through the process of individuation, the project manager combines these activities to provide the desired result. In real life, the individuation process could be a lot more complex and may involve revisions of prototypical concepts. Hence, there should be a prototypical activity, *CPU-specification*, which can be instantiated to form *CPU-Specification%1*, representing the specification development for *Micro-84 CPU version 1*. The project planner may revise the definitions as given in the prototypical activity, *CPU-specification* at the time of instantiating the activity.

Manifestations [12] are **state specific descriptions of the individuals** which describe the state at a specific time. For example, the chronicling subsystem of Callisto takes the individuated network and creates **manifestations,** which represent how and when the activities and states actually progress. For example, the manifestation for *CPU-design%1-1* will now be linked to the activity *CPU-design%1* and will provide the progress status and corresponding start and end times.

```
{{m-cpu-design%1-1
   Creation-Date: Jun 30 1983
   MANIFESTATION-OF: CPU-design%1
   MANIFESTATION-TYPE: scheduled
   STATUS: active
   DURATION: {{INSTANCE: time-interval
        START-TIME: Jan 14 1984
        END-TIME: Jan 21 1984}} }}
```

Schema 37: The manifestation of cpu-design%1

The duration slot points toward a time interval schema (to be described in V-A). There may be more than one manifestation of an activity. The manifestations are differentiated on the basis of **creation-date** and **manifestation-type.** All the scheduled manifestations are marked **scheduled,** while the real activity executions are marked manifestation-type **real.** Fig. 7 depicts these networks, where **CPU-spec%1, CPU-design%1,** and **CPU-verification%1** are the instances for the corresponding prototypical activities, *CPU-specification*, *CPU-design*, and *CPU-verification*.

Finally, let us look at the role of goals in relation to schedules and real manifestations. We view goals as a set of commitments, which change gradually with the execution of the project. There are always slips or surprises in the execution of activities, which make it difficult to predict the exact time and cost for an activity. As a result, it

[12]Thus, as in [5], instantiation is the process of linking a real thing in the world to a concept.

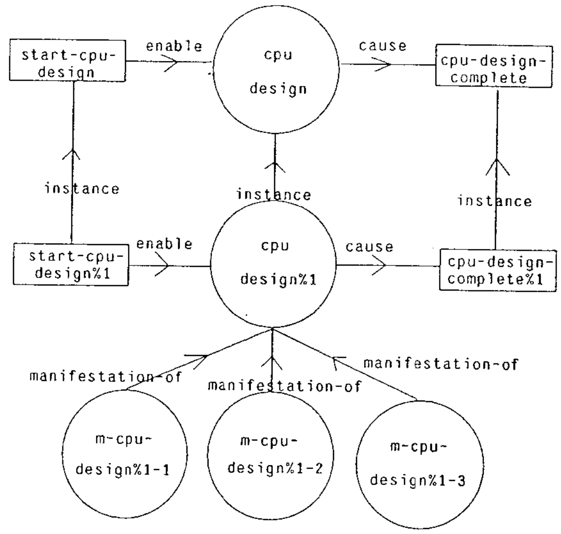


Fig. 7. Activity hierarchy and individuation.

is not uncommon to find the scheduled and real manifestations differing in values. Just from these manifestations, it is difficult to ascertain how bad a slip has been in terms of the overall goal (because the future is still unknown). The goals provide a more steady comparison point. By disaggregating the goal into subgoals or milestones, test points are created at which the status of the project can be evaluated. At each milestone, the project manager makes a decision whether to reschedule (and thus change future milestones) or not. The scheduled manifestations, on the other hand, can be changed dynamically, within a milestone, to accommodate day-to-day discrepancies in scheduled versus actual.

V. Theory of Causality and Time

Definitions of activities, states, and their abstractions only solve some of the representational problems. A manager would like to know *which conditions need to be met* before an activity starts. For example, we may assert that *CPU-design starts when CPU-specification is completed and if a CAD machine is available for the duration of the activity.* What does such an assertion mean? Does *CPU-design* start as soon as *CPU-specification* is completed? Obviously not, as we still need to check for the possession of the CAD machine. Is it sufficient, if the *CAD machine* is available at the time of starting the design and not later? Probably not, because the *CAD machine* is needed for the duration of the activity (or the activity can be suspended). We seem to understand such assertions in terms of their **temporal** and **causal** implications. It is the purpose of this section to explicitly represent this understanding of the temporal and causal implications.

Rieger and Grinberg have combined causality with temporal relations to develop a classification of cause–effect links [30]. While the resulting representation is explicit, it unnecessarily defines each cross product of causal and temporal relations (one-shot causal, one-shot enablement, continuous causal, continuous enablement, etc.). The number of such cross products increases rapidly as we begin to aggregate activities and states using logical aggregations. Also, it is not natural for us to think in t such cross products. It is a lot easier to segregate th and temporal relations and allow the model t bine any pair of them. Allen [2] has used this ap although he has not integrated time and causality w gregation across levels of detail. We have segregat and causality and have attempted to relate causal time with aggregation.

We will first define the temporal links associate the activity networks described earlier in Section We will also discuss the issues related to granula measurement of time. We will then define the cau lations which connect these concepts and show ho are abstracted to higher levels. Finally, we will issues related to separation between causality and

A. Theory of Time

"The CPU-design activity is started if the C specification is completed. The CPU-specificati should lead to a specification statement, whicl used by the design engineers for the design, otl wise one of the specification team members need accompany the design team in the design activit

While such statements are often made by projec agers, their usage or query in a model such as ou quires an understanding of the underlying tempora tions. The *completion of specification statement* a *possession of specification engineer* appear to be alt equivalent states leading to the start of the design ac While the former is a condition which needs to be " at the start of the design activity, the latter is a poss which needs to be "true" for the duration of the ac Our model of the activity should reflect the unde temporal differences in order to relate to project qu or to provide for a knowledgeable analysis of the al tives. For example, it should be possible to decide th *CPU-design activity was late because the specific were not fully generated before starting CPU-desig specification engineers could not be accessed for time due to their other commitments.*

In the modeling of activities, temporal relations pr a weak order of activities (a correlation in time as op to causality from one activity to another). For exa the activity *CPU-specification* occurs during the exec of *CPU-engineering-network*, and *CAD machine* is possessed for the duration of the *CPU-design* ity. There are three salient issues in the representati temporal information. First, there are differenes in resentation of relative and absolute time across prot ical networks and their manifestations. Second, the poral information should be abstracted across the lev activity abstraction. Last, we would like to discus issues related to measurement and comparison of tir varying granularity.

Representation of time has been a well debated [10], [7], [18], [14], [26], [1], [38], [19], and lay foundation for our work here. We will be utilizing

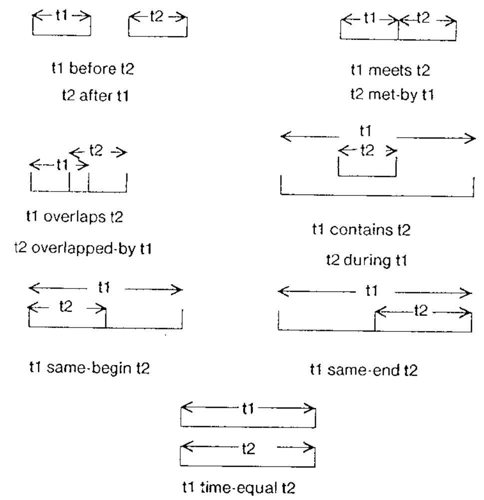


Fig. 8. The temporal relations.

temporal relations developed in [1], [38], and [19], which provide an excellent classification of temporal relationships, pictorially depicted in Fig. 8.

Each of these relations is represented as a schema with an appropriate function to resolve whether concepts follow a time relation or not [38].

```
{{before
    IS-A: temporal-relation
    DOMAIN: (or (type is-a activity)
            (type is-a state))
    RANGE: (schema (or (type is-a activity)
            (type is-a state)))
    COMPARE-FUNCTION: compare-before-fn
    INVERSE: after}}
```

Schema 38: The Before Schema

At least two concepts of time were found to occur in the representation of activities. In prototypical activity networks, the representation of time between state and activity within a cluster and between clusters was relational (e.g., *CPU-design* is done after completing *CPU-specification*). On the other hand, the temporal definitions for the manifestations of these activities are absolute (having absolute start and end times, e.g., *CPU-design starts on Feb. 15*). While the relative temporal relationships are required for the former, the latter needs a time line [38] (as illustrated below) and some way of specifying time granularity.

The first step toward the representation of time is to specify the units of time, a scale, and the functions to manipulate time. This is defined by the **time-line** schema [38]. An example of **time-line** is the *weekly-time-line*.

```
{{weekly-time-line
    INSTANCE: time-line
    POINT-FORM:
        (list (sexp (lambda (x) (not (lessp x 0))))
            (sexp (lambda (x) (not (lessp x 0))
                (not (greaterp (x 52))))))
    START-POINT: (1970 0)
    END-POINT: (1999 52)
    GRANULARITY:(0 1)
    ADD: week-add
    DIFF: week-diff}}
```

Schema 39: Illustration of Time-line

A **time interval** is defined by a schema as having a start time, an end time, and a duration. It is **dated-by** a time line. An illustration of the time-interval is *m-CPU-design%1-1 duration:*

```
{{m-cpu-design%1-1-duration
    START-TIME: (1984 2)
    END-TIME: (1984 3)
    DURATION: (0 1)
    DATED-BY: weekly-time-line}}
```

Schema 40: An Illustration of Time Interval

The slot **point-form** describes how a particular time point is represented. For example, in the *weekly time-line* schema, time is represented as a pair of year and week. The year values are restricted to positive numbers while weeks have lower bound of 0 and upper bound of 52. The **start-point** indicates the starting point of the time line (e.g., *beginning of 1970*), while **end-point** indicates the ending point of the time line (e.g., *end of the year 1999*). The **granularity** slot provides an indication of the precision of the time line. For example, in the *weekly time-line*, time durations of less than one week are ignored. The slots **add** and **diff** store the procedures to be used for adding time periods and deleting one time period from another, respectively.

1) Temporal Relations in State-Tree: First, let us describe the relational model of time in the state tree. Each relation used in the definition of the state tree has associated with it a temporal relation. These temporal associations differentiate between the one-shot precedence relations and the continuous possess relations. We will examine each of these relations specified earlier and postulate the corresponding temporal definitions.

We postulated two types of leaf states, the status predicates which model the existence of a condition, and possess predicates which model the possession of a resource. Their temporal descriptions are different. Let us define **start-time** of a state as the time in the time line at which the state becomes "true," and **end-time** as the time at which it becomes "false." The **status-predicates** are **one-shot** [30], i.e., their start time is well defined while the end time is not (only when due to a loop in the activity execution, an activity is repeatedly executed, the end time may have a meaningful interpretation). For example, the *CPU-design-complete* becomes "true" when the *CPU-design* is completed, and remains "true" unless the design is redone. On the other hand, the **possess-predicates** are **continuous** [30], i.e., both the start and the end time for the state are well defined and mark the period in time for which the state is continually "true." For example, the *CAD machine* should be possessed for the duration of the *CPU-design* activity. When a state is to be "true," it **must** be determined by the time relation explicitly linking the state, and **not** by any implicit interpretation. This implies that there should be a **meet** time relation explicitly linking the activity *CPU-design* to the state *CPU-design-complete.*

An aggregate conjunct state, composed of status-predicates, becomes "true" when all of its substates become true (see Fig. 9). The **sub-state-of** relation in such a situation is augmented with a **meet** relation in time, because the aggregate state is "true" after its substates.[13] Similarly, a composite disjunct state carries an implicit **same-**

[13]We will ignore the end-time consideration here as it is undefined.

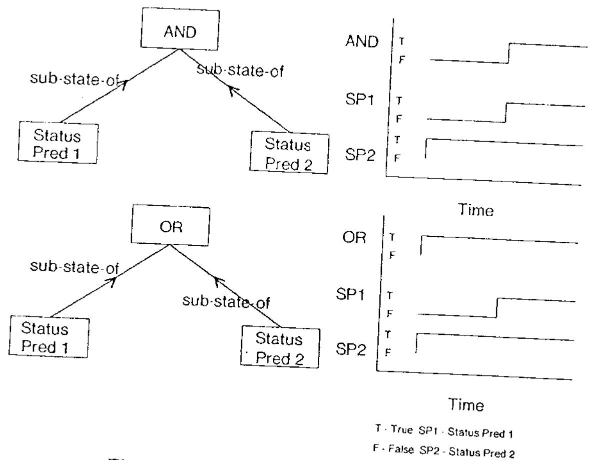


Fig. 9. Temporal aggregation of status.

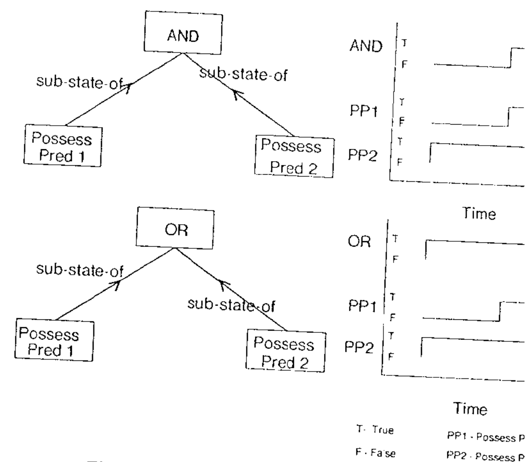


Fig. 10. Temporal aggregation of possession.

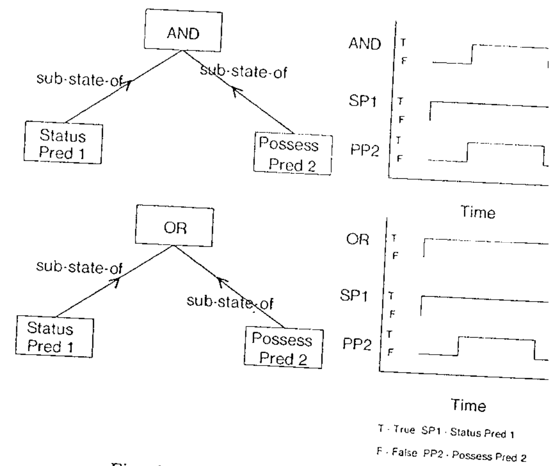


Fig. 11. Complex temporal aggregation.

begin relation, because the composite state is "true" whenever any of its substates are "true" and the two time periods have the same beginning.

An aggregate state, which has **possess-predicates** as disaggregates, is also continuous. A conjunct of possess states is "true" for the duration of time, when all of its disaggregates are "true." Unless some of them are needed for only part of the activity duration, they have an associated **same-end** relation. The disjuncts, on the other hand, have a **time-equal** relation between the aggregate and disaggregate state (see Fig. 10).

What happens now if we have an aggregate state, which is composed of both status and possess predicates? This aggregate state will have appropriate temporal relations as described above with each of the substates and will be "true" according to the complex logic created by its disaggregates. For example, if the *CPU-design activity can be stated after the specifications are listed in a report* or *if a member of the specification team can be possessed for the duration of the design activity*, then the disjunct state is **same-begin** with the status predicate (i.e., completion of specification report) and **time-equal** with the possess predicate (i.e., possession of a person to explain specifications) and the disjunct state is, then, needed to be true for the duration of the *CPU-design* activity (see Fig. 11).

There are many such alternatives and there may be any number of such composite states in a hierarchy of state tree. It is not feasible to define a complex relation for each one of these which combines a temporal characteristic with an aggregation characteristic. It is much easier to have aggregation and temporal relations coexisting in a model of the activity, so that any of these combinations can be generated and used to interpret relative temporal associations according to need. This segregation also facilitates representation of other complex temporal relations, e.g., *overlapping states* and *activities*.

2) Temporal Aggregation of Activities: The **sub-activity-of** relation carries the temporal definition of **during** because the subactivities are always done within the duration of their aggregate activity. For example, *CPU-specification* is done during the execution of the *CPU-engineering-network*. Hence, if the start or end time for specification are not given, a rough estimate can herited from the higher level activity. The definitior relation **sub-activity-of** can now be extended as fol

```
{{sub-activity-of
   IS-A: part-of during}}
```

Schema 41: The sub-activity-of relation

As opposed to the state hierarchy, the relationship tivity aggregation is consistently a temporal **durin** tion, irrespective of the type of aggregation (conju disjunct). The network in turn is an elaboration of tivity at a more abstract level and has a **time-equal** tion with the abstract activity.

3) Time Granularity: Schedule predictions and completions, on the other hand, specify absolute Consequently, the manifestations carry explicit infc tion on start and end time for the manifestation. Fo ample, the schedule for specification could specify a time of Jan. 15 and an end time of Feb. 10.

Granularity of time was defined earlier in this se as the precision of the associated time line. The gran ity of measurement needs to be defined both for **specification** and the **comparison** of temporal info tion. The specification of start or end time of a man tation implicitly contains a time granularity. For exan the statement *the CPU-design activity will be compl* *in March* uses the granularity of month (this definition of time is at a more aggregate level compared to the weekly time line we saw earlier). Similarly, the determination of whether two overlapping manifested activities occurred before, after, or during one another is dependent upon the comparison granularity. For example, if the activities *CPU-specification* and *CPU-design* are both done in the same quarter, they are **time-equal** at the granularity of a quarter, while they may meet at the granularity of a day, and may be related with an after relation at the granularity of a second.

As explained before, the concept of time line is useful in specifying the granularity, and hence two time intervals in absolute time can be compared using two different functions in different granularities. The compare function given in each temporal relation uses the granularity of the time line to adapt to the appropriate time granularity.

While being compared, the two time periods (or time points) may be specified in the same or different granularities. It is relatively easy to compare two time periods having the same granularity (e.g., the first week of March is *before* the first week of April) by using the compare function stored in the temporal relations.[14] Consider a situation where the two time intervals are specified in two different time granularities (e.g., 1984 first quarter and Feb.–Apr. 1984). One may wish to transform one time interval from one level of granularity to another before comparing and deducing the relationship (e.g., that 1984 first quarter is equivalent to Jan.–Mar. and hence overlaps with Feb.–Apr.). The question is whether such a transformation should be done on the less precise or the more precise time interval. The transformation from a more precise to a less precise time line involves an approximation, and hence, it is better to transform the time interval in quarters to the one in months and, then, apply the compare function. Time points need to be converted into time intervals before such a comparison can take place. Thus, 1984 first quarter as a time point cannot be compared to Mar. 19, until we convert it into a time period (i.e., Jan.1–Mar. 31 1984) and then, it can be deduced that 1984 first quarter contains Mar. 19, 1984.

B. Theory of Causality

In the project management domain, causation of one activity by another is central to the planning, scheduling, and chronicling of activities. We will describe here the causal primitives necessary for such a system. These causal primitives should facilitate the reasoning of causation across the activities and states. For example, someone may want to know: *Which activity is caused by CPU-specification? Which are the previous activities of CPU-design?* or *If CPU-engineering is initiated, which subactivities are started as a result?* The scheduling and chronicling systems are likely to use this causal reasoning to move through the activity network for generating a schedule or for deciphering whom to report the progress, respectively.

A number of models has been developed for tracking the progression of change or "truth" in a set of stages (e.g., Petri nets [28] and ICN [8]). Unfortunately, these models work on "flat" networks, i.e., having no aggregation or abstraction levels. We introduced aggregation and abstraction at three levels: in a state tree to describe the composite state enabling or caused by an activity, in an activity network to describe activities at different levels of detail or aggregation, and last, abstraction of states across activity hierarchy. To answer the queries raised in the previous paragraph, one needs to know the causal implications for each of these three situations. For example, it is equally correct to say that the start of *CPU-specification* causes the start of *CPU-engineering* as it is to assume the causality top-down from *CPU-engineering* to *CPU-specification* (see Fig. 12). In the project management environment, it is simply a matter of reporting versus directing, and thus, both causality directions are equally plausible.

Causality is stronger than temporal association. In the definitions of temporal relations, **meets** only specified the correlative occurrence of the two time intervals without any causation. Causation specifies an order of occurrences and has associated with it the temporal relations. In other words, temporal relations can exist without causation, while causal relations imply temporal association. Each aggregation node in the activity/state network has, associated with it, a two-way causation which needs to be used for evaluating whether a state is "true" or not. For example, if a conjunct node is "true," so are its substates. Similarly, if the substates of a conjunct state are "true," so is the conjunct state.

The three basic relations linking states to other states and activities: **enable, cause,** and **has-sub-state**, need to be formally defined to include the causality of status propagation from one state or activity to another.

Enable/Enabled-by: As when a state is linked to an activity with the **enable** relation, it introduces a new propagation-action function, so that whenever the (domain) state is true, it starts the (range) activity. For example, *CPU-design* is started when *start-CPU-design* is "**true**."

```
{{enable
    IS-A: relation
    INVERSE: enabled-by
    DOMAIN: (type is-a state)
    RANGE: (type is-a activity)
    INTRODUCTION: enable-propagation-action}}

{{enable-propagation-action
    INSTANCE: introduction-spec
    NEW-SLOT: propagation-action
    NEW-VALUE: start-activity-fn}}
```

Schema 42: The Enable relation

In the above description, the *enable-propagation-action* is introduced when the enable link is formed between the activity and its enabling state.[15] The *start-activity-fn* is a

[14] To do such a comparison, the compare function accesses the granularity and point form of the two intervals and makes an arithmetic comparison of the two tuples.

[15] For details of introduction specs, please refer to [11] and [44].

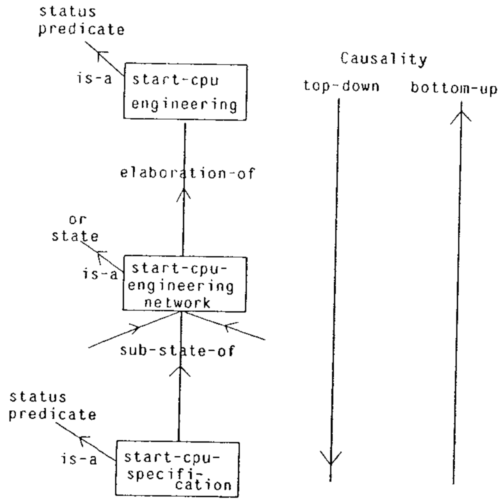


Fig. 12. Causality and abstraction.

Lisp function which is attached to the enabling state (e.g., *start-CPU-design*). Whenever a propagation action message is sent to the enabling state (à la object programming), it generates a manifestation of the activity and records the activity status as *active*.

Cause/Caused-by: The cause relation is similar to the enable relation, except that causation flows from the activity to a state. Whenever the activity changes its status, the change is propagated to the caused state. For example, the completion of *design-CPU* makes *design-complete* "**true**" (by sending a message to activate the *update-status-fn* in the **propagation-action** slot of the activity.

```
{{cause
    IS-A: relation
    INVERSE: caused-by
    DOMAIN (type is-a activity)
    RANGE: (type is-a state)
    INTRODUCTION: cause-status-action}}

{{cause-status-action
    INSTANCE: introduction-spec
    NEW-SLOT: propagation-action
    NEW-VALUE: update-status-fn}}
```

Schema 43: The Cause relation

Has-sub-state/Sub-state-of: The aggregation of states implies a two-way causality. Any changes in the status should lead to status changes in the other states in the hierarchy depending on the logical aggregation type used in the aggregation hierarchy. For example, the *or-state* in the *design-CPU* example should change its status whenever either of its disaggregate states changes in the status. The propagation-action slot is dependent on the type of aggregation and is defined for the **or-state** and the **and-state** schemata.[16]

```
{{or-state
    IS-A: aggregate-state
    PROPAGATION-ACTION: or-propagate}}
```

Schema 44: The Or-state Schema

```
{{and-state
    IS-A: aggregate-state
    PROPAGATION-ACTION: and-propagate}}
```

Schema 45: The And-state Schema

[16]The or-propagate function simply applies a Lisp "or" function to the evaluation of status by its substates, obtained by sending object programming messages. The and-propagate applies a Lisp "and" instead.

Finally, the truth propagation for the **leaf-stat** to be defined. A status predicate is **true** if it is c manifested (i.e., it has a manifestation which is and thus, has a start time but no end time), or if orations are **true.** A **possess-predicate** is **true** if i rently manifested, otherwise, a message is sent tc source manager for the required resource reque possession of the resource.

```
{{status-predicate
    IS-A: leaf-state
    PROPAGATION-ACTION: update-status-predicate-fn}}
```

Schema 46: The Status Predicate Schem

```
{{possess-predicate
    IS-A: leaf-state
    PROPAGATION-ACTION: request-resource-manager-fn}}
```

Schema 47: The Possess Predicate Schema

Truth Propagation: Are the above causal definiti ficient to define causation in activity networks? W to look at the propagation of causation from one to another to analyze whether the definitions giver lead to a nonambiguous description of the causal fl will assume that each time the system simulates th ity network to generate a schedule, the following se of steps will be executed.

1) The user initiates the **activation**[17] of the *CP neering* activity which involves sending a message state *start-CPU-engineering* to **propagation-actio** asserting that the state *start-CPU-engineering* is **tru** Fig. 13).

2) As the state *start-CPU-engineering* is a statu icate, its **propagation-action** function *update- predicate-fn* sends a message to its elaboration, *star engineering-network*, and creates a manifestation o *CPU-engineering* if the message returns a **true.**

3) The state *start-CPU-engineering-network* is **state.** The propagation-action function of an or-sta ates a manifestation of the **or-state,** if any of the me sent to its substates returns a **true.** A message is t sent to the states, *start-CPU-specification* and *start design,* for **propagation-action.**

4) The state *start-CPU-specification* is a **status- cate** without any elaborations and hence can be **true.** As a result, *start-CPU-engineering-networ start-CPU-engineering* are also made **true.** The **p gation-action** slot of *start-CPU-specification* has a tion which involves sending a message to the *CPU- fication* activity to **start.**[18]

5) The completion of the *CPU-specification* in searching for a state in the caused state tree whose matches with the status of the activity, and sending sage to that state to **propagation-action.** In this cas **propagation-action** message is sent to the state, *spec-complete*. As the state *CPU-spec-complete* is

[17]The activation of an activity implies asserting that the activity for execution. An activity can be activated either by the user, or the causation from one activity to another, as illustrated later.

[18]Starting an activity involves a number of actions: setting the s active, making a manifestation of the activity, and scheduling the completion at the scheduled end time of the activity (which is start the duration of the activity).

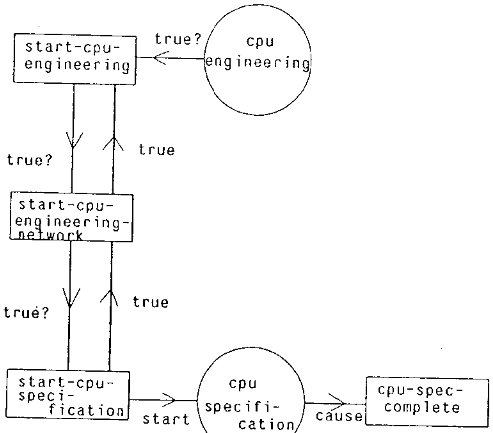


Fig. 13. The propagation of causality I.

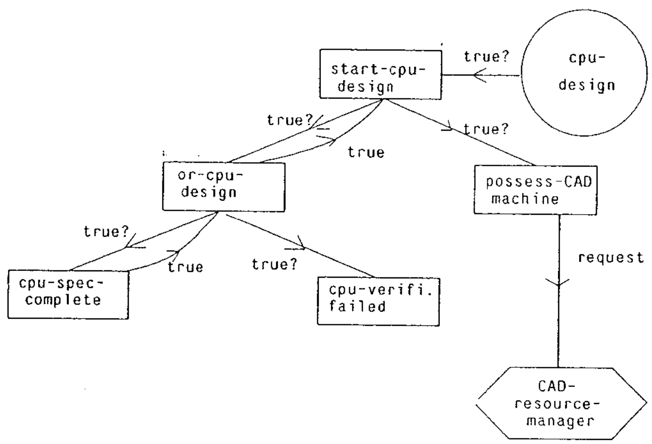


Fig. 14. The propagation of causality II.

tus-predicate with no further elaborations, a manifestation is created.

6) Now, we face a problem. In order to propagate further, we now have to move up in the enabling state tree from *CPU-spec-complete* to *start-CPU-design* (see Fig. 14). There are two ways of achieving it.

- A propagation action can be defined for moving up the state tree along the **sub-state-of** relation. But, as we have already defined a propagation action for moving down the **has-sub-state** relation, the states *CPU-spec-complete* and *or-CPU-design* will end up sending messages to each other *ad infinitum*.
- We can find the next activities of *CPU-specification* linked through *CPU-spec-complete*, activate each one of them, and follow the same logic as we did for *CPU-specification* and *CPU-engineering* activities. Following the second approach, we somehow find (to be explained, in detail, later in this section) that the activity, *CPU-design*, is to be activated. A message is sent to the state *start CPU-design* to initiate a propagation-action.

7) As *start-CPU-design* is an **and-state,** it cannot be **true** unless all of its substates are **true.** A message is sent to *or-CPU-design* and *possess-CAD-machine* to initiate propagation-action.

8) The *or-CPU-design* sends a message in turn to *CPU-spec-complete* and *CPU-verification-failed*, receives that *CPU-spec-complete* is **true** and, thus, responds in turn to *start-CPU-design* with a **true** message.

9) The state *possess-CAD-machine* is a **possess-predicate.** In order for it to be **true,** it needs to **possess** the *CAD-machine*. A message is thereby sent to the resource manager of the *CAD-machine* requesting the use of the *CAD machine* for the duration of *CPU-design* and a **false** is sent back to *start-CPU-design*.

10) When the resource manager for *CAD machine* decides to allow the possession of the *CAD machine* for the *CPU-design*, a possession message[19] is sent to the *CPU-design* and the process of propagation action is repeated for the state *start-CPU-design*.

The **truth-propagation** algorithm described above leaves one question unanswered—how are we going to find out that the activity, *CPU-design* should be activated when the activity, *CPU-specification* is completed? This question turns out to be nontrivial. Let us explore it further by defining the transitivity of a relation which moves across the state trees from one activity to its next activities.

1) The first step in such a relation is to move along a **cause** relation from an activity to the top of its **cause-state-tree.** Thus, from *CPU-specification*, we move to *CPU-spec-complete*.

2) The next step could be that of moving down a **has-sub-state** relation (e.g., from *CPU-verification-complete* to *CPU-verification-failed*, or moving up a **sub-state-of** relation (e.g., from *CPU-spec-complete* to *or-CPU-design*). As it turns out, there may be any number of such **sub-state-of** or **has-sub-state** relations.

3) Finally, one has to move from a state to an activity by moving along an **enable** relation (e.g., from *start-CPU-design* to *CPU-design*).

The problem comes from the fact that we had to move **both** up and down the state trees. There is no consistent way of ensuring that we stop at only the next activities of the activity that we started with. Let us consider the situation in Fig. 15. We would like to model a network where activity *a*1 has two alternative outcomes: *s*121 and *s*122. Another activity *a*2 has a single outcome *s*22. Activity *a*3 starts when *a*1 causes *s*122 or *a*2 causes its completion, *s*22. Also, activity *a*4 starts after *a*2 results in *s*22. There is nothing to inform the "transitivity" algorithm, which attempts to move from *s*12 to *s*122, that after having moved to *s*31, it should not move to *s*22. This portrayal of state trees does not have an associated concept of causality, which would have differentiated between movement from *s*122 to *s*31 and from *s*31 to *s*22.

Let us diagnose the problem a little more closely. Each status predicate in our network stands for two descriptions. First, that a condition is met due to the ending of an activity (e.g., *failure of verification at the end of the verification activity*). Second, that this condition is one of

[19] Similar to the activation message, a possession message informs the activity that the needed resource is available and that the activity can be started.

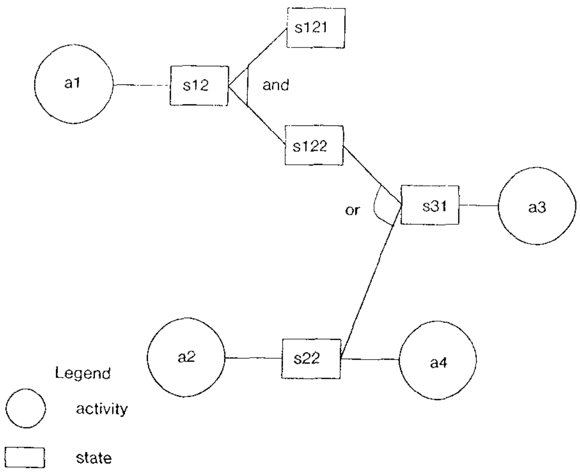


Fig. 15. Causation: problems in transitivity.

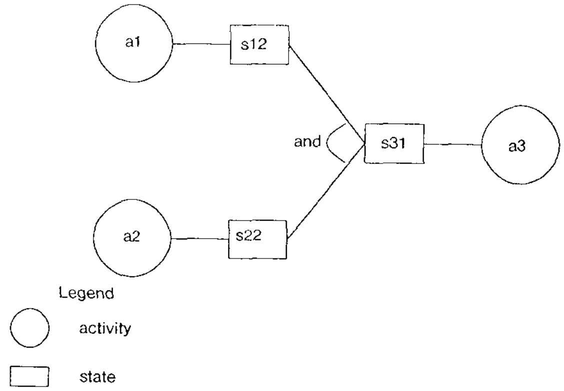


Fig. 16. Causation: state space approach.

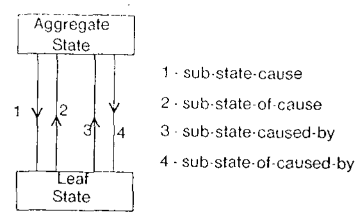


Fig. 17. Causation: multiple link alternative.

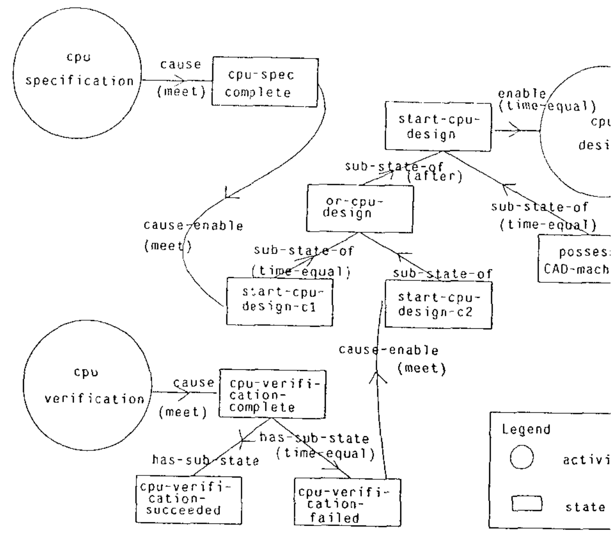


Fig. 18. Activity clusters illustration.

the states required for starting a new activity (e.g., *starting design due to failure of verification*). It is tempting to use one state to signify both of the above, as it is done in the state space approach [28], [8]. While modeling simple activity networks, these two states naturally collapsed together without adding any ambiguity to the definition of next-activities. In Fig. 16, the flow of causality moves from activity $a1$ to state $s12$, from activity $a2$ to state $s22$, then from these two states to their conjunct $s31$, and finally, from state $s31$ to activity $a3$. Here, the same state $s12$ signified the completion of activity $a1$ and a condition for starting activity $a3$. We should be able to model this network with $s31$ as a conjunct state made of two leaf states $s12$ and $s22$.

Aggregation among activities and states introduces ambiguity. As we saw in Fig. 15, the state space approach is clearly inadequate for dealing with arbitrarily complex activity and state combinations.

One way of dealing with the problem of transitivity of causality and "truth propagation" is to define two causal links for each substate/subactivity relation. Each state in such a network would be a part of a causal chain at a level of abstraction, and there would be additional links to relate to higher levels of abstraction. Fig. 17 shows an illustration of this approach. Although this approach is explicit, it ignores the implications from the semantics of aggregation and abstraction relations and increases necessarily) the number of causal relations.

We see a need for separation of causality from a; gation. This involves the modeling of causality separ across the **activity clusters** (see Fig. 18) (defined e in Section IV-A) and using the state trees for ascerta causality within an activity cluster. It looks reasonat follow this approach because activity cluster is an a; gate concept capable of reasoning within itself to pr the direction of causality. We use the relation **caus able** to link the caused state associated with one ac to the corresponding enabling state of its next act Hence, *CPU-spec-completion*, the caused state assoc with specification, has a cause-enable link to *start- design c1*. Within the activity cluster for *CPU-desigr* transitivity algorithm can move along the state agg tion from the leaf states to the top of the tree. The tration in Fig. 15 now changes to Fig. 19.

This approach involves the definition of the state *s* as a mirror image of $s122$, which is **true** wheneve latter is true. An added benefit of the inclusion of act clusters is that they reduce the complexity of activit iting. Activities are modular, and hence are easier to i and modify without worrying about interactions with states.

The **cause-enable** relation was introduced to sep clusters. The link is not only used to separate but al propagate state changes. For example, when *CPU-ve cation* fails, the **cause-enable** relation, which conr *CPU-verification* activity cluster to *CPU-design*, pr gates a change in status and, thus, initiates the sta

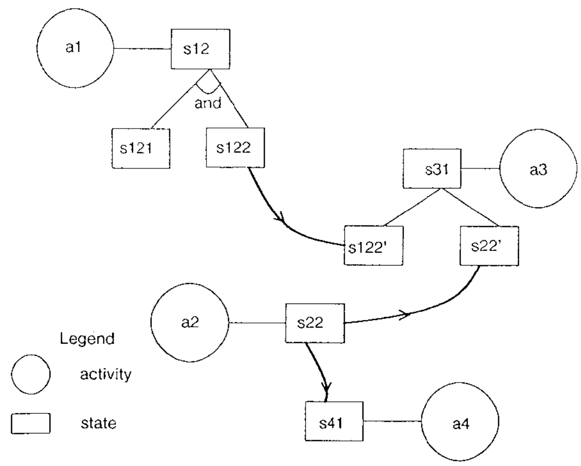


Fig. 19. Causation: aggregations and cause-enable.

CPU-design. The enabling state tree for *CPU-design* represents the other requirements for starting specification, i.e., having the CAD machine. Once the CAD machine is available, it fulfills the condition for starting *CPU-design* and propagates the causation to the *CPU-design* activity itself by forming a manifestation of the *CPU-design* with an appropriate start time.

Cause-enable/Enable-cause: The **cause-enable** relation allows propagation of status from one state to another. Whenever the domain state changes its status, the range state should also change its status accordingly.

```
{{cause-enable
    IS-A: relation
    INVERSE: enable-cause
    DOMAIN (type is-a state)
    RANGE: (type is-a state)
    INTRODUCTION: cause-enable-propagation-action}}

{{cause-enable-propagation-action
    INSTANCE: introduction-spec
    NEW-SLOT: propagation-action
    NEW-VALUE: create-manifestation-fn}}
```

Schema 48: The Cause-enable relation

C. Time, Causality, and Goals

Having defined time and causality, let us look at their association. We have found that causality is a stronger association of the two because it implies the temporal association as well as the direction of causation. At the same time, there are a number of combinations generated from the association of time, causality, and aggregation and we do not have a small set of combinations which could be labeled and used together. We would like to raise here two issues related to time, causality, and aggregation. First, how are the temporal and the causal links related? Second, what is the role of goals and milestones?

We have employed many of the causal relations given in [30], but have described time separately. This separation enables activities to be causally linked with a temporal relation ascribed separately. In general, there is no need to assign a specific temporal link as the process of causation (i.e., the "truth" propagation) will generate a temporal association in absolute time and a similar system can simulate the causal reasoning to derive the relative temporal associations. For example, if no temporal associations are provided, the system can reason through the network to assume that *specification is completed before starting design*. The temporal relations at the same time provide additional information not available in the causation (e.g., *specification can overlap with design*).

The **goal-states** defined earlier imply a need for causation from an initial state to a goal state. The question is: what do these goals imply in terms of time and causality? In particular, the **must-satisfy** relation projects what should happen. What does this **must-satisfy** relation imply in terms of causality, and how should the "truth-propagation" deal with the **must-satisfy** relation, and finally, which of the time relations should be associated with the **must-satisfy** relation?

The status of goals is different from "true" and "false." A goal is either **inactive, active,** or **satisfied.** When the goal is generated, it is **inactive** as no activity is actively pursuing the achievement of the conditions specified in the goal. The enablement of the attached activity leads to a change in the goal state from **inactive** to **active.** The active status of the goal implies that an activity is being pursued to meet the goal. If the completion of the activity meets the conditions specified in the goal, it **satisfies** the goal. The goal state is manifested for each of the three above mentioned states by the respective actions, viz., enablement of the enabling state tree and causation of the caused state tree. The manifestations carry a status value and a time interval during which the goal was in the specified status. Thus, the goal for version 1 of *CPU-engineering%1* is set to active as and when the *start-CPU-engineering%1* activity is manifested. The goal is satisfied when the *CPU-engineering%1* is completed (see Fig. 20).

VI. Theory of Relational Abstraction

When we walked into the application environment, the first couple of sessions were spent in understanding the meaning of words used. Terms like ECO, revisions, components, etc., had specific meanings associated. Once we generated the semantic representation, we had to generate abstract relations to conform to the managers' vocabulary.

It is interesting to note that while the jargon seemed obtrusive to people outside the organization, it was used freely within the organization, with no ambiguities. When we examined the meaning, we could theorize and represent the underlying semantics. Organizations develop their own languages and everyone communicates in these languages, despite the fact that the language used will not be understood by outsiders. In this section, we will discuss the rationale behind these domain languages and the relaated issues of **representational complexity** and **distance.** Unfortunately, the underlying semantic representation is usually available in the minds to the system designers alone. In our representation, it is possible to overlay the domain structure over the semantic structure and change the domain layer from organization to organization or from one application to another.

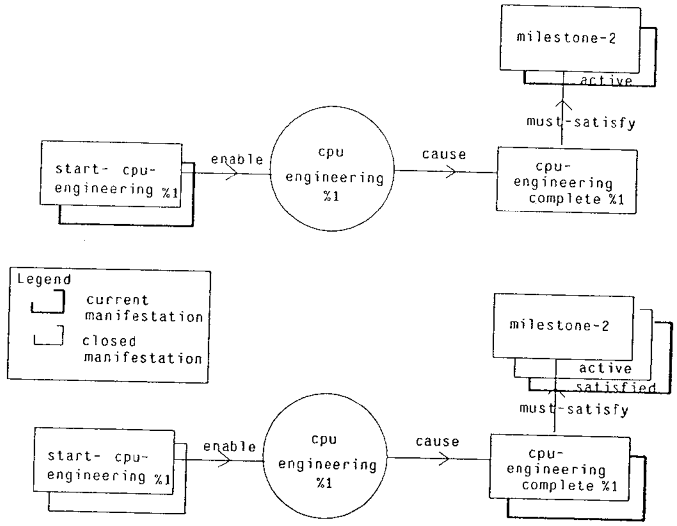


Fig. 20. The status of goal.

Another reason for overlaying abstract relations is the complexity of the representation. While the explicit representation of time, causality, etc., is theoretically satisfying, in practice it places a heavy burden on the creator of the model. It is obvious to the model builder how activity clusters are formed and traversed, but in applying these concepts for perusing the database, the model builder or the user would like to use more abstract relations. The problem here is one of *representational distance*, that is, how complex is the transformation of surface-level concepts into the representation primitives. Frame systems provide a partial solution similar to abstract data types; a frame represents an aggregation of properties and structure. The SRL language used by Callisto also provides relation abstraction. Hence, "higher level" relations may be defined.

For example, let us develop a relation *next-activity-of* which links two activities, causally linked to each other. We would like to infer that *CPU-design* is the **next-activity-of** *CPU-specifiation*. In the model developed in Section IV, we described *CPU-specification causes CPU-spec-complete, which has a cause-enable link to start-CPU-design-c1, which in turn is the substate of or-CPU-design. The state or-CPU-design is sub-state of start-CPU-design, which enables the activity CPU-design*. The relation **next-activity-of** is an abstraction of this detailed description. The relationship between the abstract relation and its elaboration are provided by defining the transitivity of **next-activity-of** in terms of the basic relations used at the semantic level. The schema representation of **next-activity-of** is as follows:

```
{{next-activity-of
    IS-A: relation
    INVERSE: has-next-activity
    DOMAIN: (type is-a activity)
    RANGE: (type is-a activity)
    TRANSITIVITY:
    (list
        (step enabled-by all t)
        (repeat (step has-sub-state all t) 0 inf)
        (step enable-cause all t)
        (repeat (step sub-state-of all t) 0 inf)
        (step caused-by all t))}}
```

Schema 49: The Relation next-activity-of

The user could query whether the activity *CPU-d* is **next-activity-of** *CPU-specification*, or could get of activities, each of which are **next-activity-of** *specification*.[20]

Linguistic level [5] relations may be formed with combination at the conceptual or epistemological le Hence, if the project manager intends to specify **sub eration-of** as a **sub-activity-of** for the manufacturing main, he should be able to specify it by defining the **operation-of** relation as follows:

```
{{sub-operation-of
    IS-A: sub-activity-of
    DOMAIN: (type is-a operation)
    RANGE: (type is-a operation)
    INVERSE: has-sub-operation
    TRANSITIVITY: (step sub-activity-of all t)}}
```

Schema 50: The Sub-operation-of Schema

To summarize, the purpose of relational abstractio twofold. First, it provides a way of representing relat which are abstractions of detailed semantic represe tion. This abstract representation reduces the sema complexity that the model builder or the user has to with. Second, it helps translation of domain concept more general semantic concepts.

VII. Conclusion

We started with a goal of developing a representat language which satisfies the criteria of completeness, p cision, and lack of ambiguity. In the process of develop the representation language, we integrated the theorie activity, time, causality, manifestation, and instantiati The integration process raised a number of issues: fi the need for separation of time and causality; second, difference between one-way causation (the **cause-ena** relation) and two-way causation (the **has-sub-state** a **sub-state-of** relations); and finally, the difference betwe the aggregation process of building a whole from its par from the abstraction process of reducing information fr one level of detail to another.

A formal evaluation of these theories is a paper in itse We will concentrate here on the example listed in Secti II, and evaluate the theories in terms of their complet ness, precision, and lack of ambiguity.

*Evaluation of **Completeness:*** The criterion of co pleteness requires that the representation spans the app cation domain. The project management tasks ne models of the required activities; their duration; prec dence, resources; time; logical (causal) connections; i dividual and prototypical plans; constraints and orga zations for conflict resolution. This paper includes t definition of the activities, the states or conditions en bling the activity, and those caused by the activity. It co

[20]In the representation of this relation, we have specified the followi transitivity grammar for the relation. In order to relate two activities usi this relation, one has to traverse one **enabled-by** relation, any number **sub-state relations** (which take us down the enabling state tree), one e**able-cause** relation (which takes us to the caused state tree of the oth activity), any number of **sub-state-of** relations to go up the caused sta hierarchy, and finally a **caused-by** relation to reach the activity.

ers the activity precedence and resource requirements, individual and prototypical plans, and alternative manifestations, as well as the temporal and causal relations linking these activities and states. The concepts related to project environment, i.e., the oganization for conflict resolution, are described elsewhere [35]. The theory of constraint can be found in [12]. At the same time, the theory falls short in the description of activity attributes (e.g., cost, duration, product, or state transformation details), the procedures for aggregation and abstraction of activities and states (e.g., the operations needed for aggregation-averaging, summation, etc., and the types of attributes for each of these operations), and the use of classification relations for categorizing and generating group characteristics.

Evaluation of ***Precision:*** Precision requires description to be at the appropriate granularity of knowledge, i.e., the precision used in the project management communication. The theory is considered successful if the sentences in the example can be translated into a set of concepts which replicate the descriptions in the sentences. Using relational abstraction, a number of higher level statements can be faithfully replicated (e.g., *specification is followed by design*). At the same time, the theory is capable of describing the situation in a lot more detail (e.g., *what conditions need to be met before the CPU-design activity? or during the CPU-design activity?*). Thus, a user can choose the appropriate level of precision in describing plans, schedules, or progress in a project.

Evaluation of ***Clarity:*** Clarity of the theories can be evaluated by ensuring that there exists one and only one representation for a given situation. These are two likely sources of ambiguity: **inconsistency** and **incompleteness** (of which completeness is covered above).

Inconsistency implies that there exist two or more project descriptions which, when put together, give rise to a conflict. For example, if managers use different PERT-based networks for project descriptions at different levels of the managerial hierarchy, the descriptions may suffer a lack of common updating procedures. Similar problems have been observed during plan generation and scheduling of projects. We aimed at providing **explicit details** not only to avoid **incompleteness** but also to achieve the **integration** of concepts so as to avoid **inconsistency.** For example, the integration of *CPU-engineering* activity with *CPU-engineering-network* ensures that the status information remains consistent between the two levels of detail. The specifications and changes made in planning, scheduling, or chronicling are integrated at multiple levels in the managerial and project hierarchy, not only across levels of management, but also within a level from one department or unit to another. In this way, the introduction of inconsistency is minimized and inconsistencies that do exist are brought to the surface.

Research tends to raise as many questions as it answers. Our work is no different. It raises issues in two directions.

- Whether the experimental system developed here can be applied to "real-life" large engineering and manufacturing projects. A number of questions is often asked. For example, how much detail is really needed? How easy will it be to use? How bulky will it be? Would it be adequate for all project management needs? While such large projects involve 5000 or more activities, no manager ever reviews more than 100 activities at a time. The major shortcoming of the existing commercial packages is their inability in summarizing or focusing on the 100 relevant activities. While our research paves the way, the techniques for presenting summaries and focuses are yet to evolve.
- The activity representation is similar across the various application domains. While we developed a set of semantic primitives, they need to be validated on a large number of domains. It would be worthwhile to explore the similarities and differences across domains, especially in their inheritance considerations.

Acknowledgment

We would like to acknowledge the helpful comments from L. C. Smith, W. Sears, G. Mangan, R. Glackemeyer, S. Smith, E. Screven, and P. Gage.

References

[1] J. F. Allen, "Maintaining knowledge about temporal intervals," *Commun. Assoc. Comput. Machine*, vol. 26, pp. 832–843, Nov. 1983.

[2] ——, "General theory of action and time," *Artif. Intell.*, vol. 23, no. 2, July, 1984.

[3] D. Bobrow and T. Winograd, "KRL: Knowledge representation language," *Cogn. Sci.*, vol. 1, no. 1, 1977.

[4] R. J. Brachman, "A structural paradigm for representing knowledge," Ph.D. dissertation, Harvard Univ., Cambridge, MA, May 1977.

[5] ——, "On the epistemological status of semantic networks," in *Associative Networks: Representation and Use of Knowledge by Computers*, N. V. Findler, Ed. New York: Academic, 1979, pp. 3–50.

[6] ——, "What is-a and isn't: An analysis of taxonomic links in semantic networks," *IEEE Trans. Comput.*, vol. C-32, pp. 30–36, Oct. 1983.

[7] B. C. Bruce, "A model for temporal references and its application in a question answering program," *Artif. Intell.*, vol. 3, pp. 1–25, 1972.

[8] C. Ellis, "Information control nets: A mathematical model of office information flow," in *Proc. ACM Conf. Simulation Meas. Model. Comput. Sys.*, 1979.

[9] S. E. Fahlman, "A system for representing real world knowledge," Ph.D. dissertation, Mass. Inst. Technol., Cambridge, MA, 1977.

[10] N. V. Findler and D. Chen, "On the problems of time, retrieval of temporal relations, causality, and co-existence," in *Proc. 2nd. Int. Joint Conf. Artif. Intell.*, 1971, pp. 531–545.

[11] M. S. Fox, "On inheritance in knowledge representation," in *Proc. 6th Int. Joint Conf. Artif. Intell.*, Tokyo, Japan, 1979.

[12] ——, "Constraint directed search: A case study of job-shop scheduling," Ph.D. dissertation, Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, 1983.

[13] I. Goldstein and B. Roberts, "NUDGE, a knowledge-based scheduling system," in *Proc. 5th Int. Joint Conf. Artif. Intell.*, 1977, pp. 257–263.

[14] P. J. Hayes, "The naive physics manifesto," in *Expert Systems in the Micro Electronic Age*, D. Michie, Ed. Scotland: Edinburgh University Press, 1979, pp. 243–270.

[15] G. G. Hendrix, "Modeling simultaneous actions and continuous processes," *Artif. Intell.*, vol. 4, no. 3, pp. 145–180, 1973.

[16] ——, "Expanding the utility of semantic networks through partitioning," presented at the 4th Int. Joint Conf. Artif. Intell., Tiblishi, U.S.S.R., 1975.

[17] ——, "Encoding knowledge in partial networks," in *Associative Networks, Representation and Use of Knowledge by Computers*, N. V. Findler, Ed. New York: Academic, 1979.

[18] K. M. Kahn and A. G. Gory, "Mechanizing temporal knowledge," *Artif. Intell.*, vol. 9, no. 2, pp. 87–108, 1977.

[19] B. I. Kedzierski, "Knowledge-based communication and management

and support in a system development environment," Ph.D. dissertation, Dep. Comput. Sci., Univ. S. Western Louisiana, Lafayette, LA, Nov. 1983. See also, Kestrel Inst., Palo Alto, CA, Tech. Rep. KES.U.83.3

[20] J. E. Kelley and M. R. Walker, "Critical-path planning and scheduling," in *Proc. Eastern Joint Comput. Conf.*, 1959.

[21] R. M. Lee, "CANDID: A logical calculus for describing financial contracts," Ph.D. dissertation, Dep. Decision Sci., Wharton School, Univ. Pennsylvania, Philadelphia, PA, 1980.

[22] D. Lenet, "AM: An artificial intelligence approach to discovery in mathematics as heuristic search," Ph.D. dissertation, Dep. Comput. Sci., Stanford Univ., Palo Alto, CA, 1976.

[23] F. K. Levy, G. L. Thompson, and J. D. Wiest, "The ABC of the critical path method," *Harvard Business Rev.*, Oct. 1963.

[24] D. G. Malcolm, J. H. Rosenbloom, and C. E. Clark, "Application of a technique for research and development program evaluaton," *Oper. Res.*, Sept.-Oct. 1959.

[25] J. McCarthy, "Situations, actions and causal laws," Stanford Univ., Palo Alto, CA, Tech. Rep. AIM-2, July 1963.

[26] D. McDermott, "A temporal logic for reasoning about processes and plans," *Cogn. Sci.*, vol. 6, pp. 101-105, 1982.

[27] J. R. Meehan, "Everything you always wanted to know about authority structures but were unable to represent," presented at the 1st Nat. Conf. Artif. Intell., 1980.

[28] J. L. Peterson, "Petri nets," *Comput. Surveys*, vol. 9, no. 3, pp. 224-252, Sept. 1977.

[29] M. R. Quillian, "Semantic memory," Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, PA, 1966.

[30] C. Rieger and M. Grinberg, "The declarative representation and procedural simulation of causality in physical mechanisms," in *Proc. 5th Int. Joint Conf. Artif. Intell.*, 1977, pp. 250-255.

[31] R. B. Roberts, and I. P. Goldstein, "The FRL primer," Artif. Intell. Lab., Mass. Inst. Technol., Cambridge, MA., Tech. Rep. Memo 408, 1977.

[32] E. D. Sacerdoti, "Planning in a hierarchy of abstract spaces," in *Proc. 3rd Int. Joint Conf. Artif. Intell.*, 1973, pp. 412-422.

[33] ——, "Planning in a hierarchy of abstract spaces," *Artif. Intell.*, vol. 5, no. 2, pp. 115-135, 1974.

[34] A. Sathi, M. S. Fox, M. Greenberg, and T. Morton, "Callisto: An intelligent project management system—Overview," Intell. Syst. Lab., Robot. Inst., Carnegie-Mellon Univ., Pittsburgh, PA, 1985.

[35] ——, "Modelling of project environment," Intell. Syst. Lab., Robot. Inst., Carnegie-Mellon Univ., Pittsburgh, PA, in preparation.

[36] R. Schank and R. Abelson, *Scrips, Plans, Goals and Understanding.* Hillsdale, NJ: Lawrence Erlbaum, 1977.

[37] L. K. Schubert, "Extending the expressive power of semantic networks," *Artif. Intell.*, vol. 7, pp. 163-168, 1976.

[38] S. F. Smith, "Exploiting temporal knowledge to organize constraints," Intell. Syst. Lab., Robot. Inst., Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-RI-TR-83-12, 1983.

[39] M. Stefik, "An examination of a frame-structured representation system," in *Proc. 6th Int. Joint Conf. Artif. Intell.*, Tokyo, Japan, 1979.

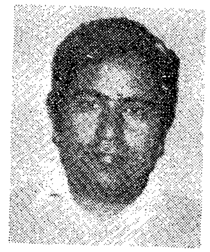
[40] A. Tate, "Generating project networks," in *Proc. 5th Int. Joint Conf. Artif. Intell.*, 1977, pp. 888-893.

[41] E. Turban, "The line of balance—A management by exception in *Project Management: Techniques, Applications and Manage sues*, E. W. Davis, Ed. New York: American Institute of Ind Engineers, 1976, pp. 39-47.

[42] A. M. Webster, *Webster's Ninth New Collegiate Dicti* Springfield, MA: Merriam Webster, 1983.

[43] W. A. Woods, "What's in a link: Foundations for semanti works," in *Representation and Understanding*, D. Babrow a Collins, Eds. New York: Academic, 1975.

[44] J. M. Wright, M. S. Fox, and D. Adam, *SRL/2 Users Manual*, I Inst., Carnegie-Mellon Univ., Pittsburgh, PA, 1984.



Arvind Sathi received the B.S. degree in e cal engineering from the Indian Institute of nology, Kanpur, India, in 1977, the M.B.A gree from the Indian Institute of Manage Bangalore, India, in 1979, and the M.S. deg marketing and systems sciences from Carn Mellon University, Pittsburgh, PA, in 1983.

He expects to receive the Ph.D. degree in from GSIA, Carnegie-Mellon University i area of distributed project management. He i rently a Senior Scientist at Carnegie Group, Pittsburgh, PA, working in the area of knowledge-based system ap tions to project and production management. Prior to joining Car Group, he worked on the Callisto project (an intelligent project mar ment system sponsored by Digital Equipment Corporation) at the Rob Institute of Carnegie-Mellon University.

Mark S. Fox (S'76-M'79-S'79-S'80-M'80-M'81) for photograph an ography see p. 501, this issue.

Michael Greenberg received the B.S. degree electrical engineering from Carnegie-Mellon U versity, Pittsburgh, PA, in 1982.

He is currently a doctoral candidate at the U versity of Massachusetts, Amherst. He a worked on Expert System development at the botics Institute of Carnegie-Mellon University on the Callisto project at Digital Equipment C poration. His current research is in the area of r soning about uncertainty.