Eliciting Software Process Models with the E^3 Language

Maria Letizia Jaccheri
Norwegian University of Science and Technology (NTNU)
and
Gian Pietro Picco and Patricia Lago
Politecnico di Torino, Italy

Software processes are complex entities that demand careful understanding and improvement as they determine the quality of the resulting product. A necessary step towards the improvement of an organization's process is a clear description of the entities involved and of their mutual relationships.

Process model *elicitation* aims at constructing this description under the shape of a software process model. The model is constructed by gathering, from several sources, process information which is often incomplete, inconsistent, and ambiguous. A process modeling language can be used to represent the model being elicited. However, elicitation requires process models to be understandable and well structured. These requirements are often not satisfied by available process modeling languages because of their bias towards process enaction rather than process description.

This paper presents a process modeling language and a support tool which are conceived especially for process model elicitation. The E^3 language is an object-oriented modeling language with a graphical notation. In E^3, associations are a means to express constraints and facilitate reuse. The E^3p-draw tool supports the creation and management of E^3 models and provides a view mechanism that enables inspection of models according to different perspectives.

Categories and Subject Descriptors: D.1.5 [**Programming Techniques**]: Object-oriented programming; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.2 [**Software Engineering**]: Tools and Techniques—*Computer-aided software engineering (CASE)*; D.2.9 [**Software Engineering**]: Management; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*Software development, Software maintenance*

General Terms: Software Process

Additional Key Words and Phrases: Software process modeling, process model elicitation, associations

---

Authors' addresses: M.L. Jaccheri, Norwegian University of Science and Technology (NTNU), N-7034 Trondheim, Norway; e-mail: `letizia@idi.ntnu.no`; G.P. Picco and P. Lago, Dipartimento di Automatica e Informatica, Politecnico di Torino, C.so Duca degli Abruzzi 24, I-10129 Torino, Italy; e-mail: `{picco|patricia}@polito.it`. All correspondence should be directed to the first author.

1. INTRODUCTION

Software processes are complex activities that affect critical parameters such as final product quality and costs. A considerable research effort is currently spent in academia and industry to understand how to improve existing software processes.

Some of the approaches address process improvement from a managerial perspective: this is the case of the Capability Maturity Model [Paulk et al. 1993], the Bootstrap project [Bootstrap Project Team 1993], and the Quality Improvement Paradigm [Basili and Rombach 1988]. These approaches stress the importance of characterizing the process in order to assess it, evaluate its strengths and weaknesses, and identify an action list for improvement. Although a process description is highly recommended, these approaches emphasize evaluation and improvement of the process rather than its description.

Other approaches adopt a technological perspective: the research area known as *software process modeling* [Finkelstein et al. 1994] can be classified under this umbrella. A process model is a specification of a real world software process. By analogy to software system specification [Ghezzi et al. 1991], a specification aims at describing precisely and unambiguously the requirements, the design, or the implementation. Hence, a process model can have different purposes depending on the meta-process[1] activity that produces it. Huff identifies the main goals of software process modeling as understanding, improvement, and *enaction*[2], and points out the importance of analysis of software process models in achieving any of the goals above [Huff 1996]. However, the process modeling community traditionally [Taylor et al. 1988; Kaiser and Feiler 1987] has focused on the ultimate goal of supporting enaction of process activities within Process-centred Software Engineering Environments (PSEEs). In this context, the process model provides the description for the activities to be executed. As a consequence, the majority of the existing Process Modeling Languages (PMLs) have privileged the language aspects that enable efficient and sophisticated execution mechanisms in a PSEE, such as reflection, and tool and data integration. This has led to PMLs that are often non-intuitive and hence difficult to exploit for process elicitation, where communication and understanding of the resulting process models is of paramount importance for users, who are not necessarily computer scientists. For these reasons, the exploitation of these PMLs in process improvement initiatives is limited.

The process modeling community is currently debating whether PSEEs can be adopted effectively by real organizations. This seems to require technological, organizational, and cultural endeavors that are too demanding for the majority of software factories. Moreover, a precise assessment of the economical benefits provided by process model enaction is still missing.

Nonetheless, process modeling already provides a suitable technology to build the process model mandatory for guiding any process-related activity, concerning both development and improvement. The importance of process modeling for elicitation rather than for enaction has been recognized already [Humphrey and Kellner 1989;

[1]The process of creating process models [Dowson et al. 1991; Conradi et al. 1992; Finkelstein et al. 1994].

[2]The term *enaction* is commonly used in the process modeling field to denote process execution carried out jointly by human and automated process actors.

Kellner and Hansen 1989]. Moreover, recent case studies [Bandinelli et al. 1995; McGowan and Bohner 1993; Aumaitre et al. 1994; Barghouti et al. 1995] have evidenced that the elicitation of a process model can be useful *per se* even when the goal is not an enactable model, but rather the creation of a process description to be used in the context of a general process improvement strategy. These case studies describe the elicitation and specification of the process model of real world organizations. They exploit a notation or a language not only for description, but also to help communication and interaction between the modeler and the process user during the elicitation activity and the related verification. However, the languages and notations used were conceived either for specification of software systems or for process enaction. As discussed in Section 1.1, process model elicitation poses special requirements and goals [Huff 1996].

In summary, the elicitation activity should be regarded as the *trait d'union* between the managerial and technological approaches to software process improvement. Our goal is to support process model elicitation activities by providing a new formal PML, called E^3 PML, that is designed[3] with this goal in mind and without the initial commitment toward process enaction.

Given the focus on process improvement, the major requirements for our PML are simplicity combined together with modularization and abstraction facilities. Simplicity is needed because typical process users are not computer scientists, and therefore concentrate on how to model the process precisely rather than on the nuts and bolts of the modeling language. Modularization and abstraction are needed to cope with modeling in the large and to foster reuse of process model fragments. Finally, we require our language to be formal, to allow verification of properties and leave room for further developments in the area of simulation and, possibly, enaction.

1.1 Software Process Model Elicitation

If a PML has to be used for eliciting a software process, its design is constrained in several ways. In this section, we describe the distinctive characteristics of the elicitation meta-process activity and sketch the requirements we used as a basis for the design of our language.

Elicitation of a process model does not include any description of automation or enaction choices. The goal of elicitation is to understand a process and describe it gradually by means of a model. The elicitation of the software process model of an organization is often the very first attempt to define it, trying to build a description that is as close as possible to how the *real* process is carried out [Bandinelli et al. 1995]. Consequently, during this activity, process information is gathered from several sources which is often incomplete, inconsistent, and ambiguous. Common sources include documentation (e.g., quality manuals written in natural language), process owners, and process performers[4].

Meta-processes for the elicitation activity, involving the explicit use of process

[3] E^3: Environment for Experimenting and Evolving software processes.

[4] Hereafter, we use the term *process user* to denote both a process owner and a process performer, i.e. whoever makes use of a process model, and *process modeler* to denote the role in charge of developing software process models.

models, are discussed in the aforementioned case studies. During elicitation, the modeler describes the information gathered by means of model fragments. The fragments are refined gradually, both to fulfill requirements of the process users and to be internally consistent. Modeling is usually interleaved with verification performed jointly by the process users and the modeler. Finally, model fragments are integrated to derive the final, fully formalized model of the software process. Since, fragments modeling different parts of the process may be derived from inconsistent sources of information, they may be reconciled and modified to restore a globally consistent model. Multiple integration sessions should be interleaved with verification of model consistency and completeness. Model verification is achieved through analysis, which can be performed either on the static representation of the process or on its dynamic behavior, as prescribed by the model. Static analysis mechanisms may encompass syntactic and semantic checks [Deiters and Grun 1994], as well as queries and views on the model [Curtis et al. 1992; Tanaka et al. 1995]. On the other hand, dynamic analysis techniques rely on either simulated [Mi and Scacchi 1990] or real [Cook and Wolf 1994] enaction of the process model. In the latter case, analysis is performed on process data collected during enaction.

We propose the use of a single formal language across the aforementioned activities, thus encompassing description, verification, and integration. A formal language with precise syntax and semantics helps in reducing the ambiguity of the model fragments being communicated. Also, a formal language with modularization and abstraction facilitates the process of producing a formally structured model and helps in *reusing* the fragments created during elicitation. Our language has been designed around three requirements for process model elicitation:

Model understanding and domain coverage. As we already pointed out, an understanding of the model is of paramount importance for elicitation. If formality is introduced, this should not hamper understanding. The language must also provide features both to map entities directly from the real world into model components and to model the constraints present in the real world as constraints among model components. Real world entities include activities, documents, software tools, and resources. Constraints can be interaction patterns among activities, such as parallelism and non-determinism, document sharing policies, and so on. Furthermore, a graphical notation for the formal language is highly desirable, since the effectiveness of visual representation has been proven in many fields of software engineering.

Model structure and reuse. Paraphrasing Krueger [Krueger 1992], reuse in process modeling can be defined as the activity of using existing software process models when developing a new model. Reuse depends on the *structure* of the existing model, i.e. on the quality and abstraction level of the model components, and on the way they are assembled. Krueger also emphasizes that a model is well structured when its components can be reused. To fulfill the requirements of structure and reuse, language level abstraction facilities like inheritance or modularity, as well as multiple view mechanisms are needed. Reuse should be achieved along two dimensions. First, each process model consists of primary elements like tasks, roles, tools, and software products, together with the relationships among them. Thus, a PML should offer these primary process elements as first class elements, just like characters, strings, and integers are offered by a programming language. This achieves the dual goal of avoiding unnecessary definition of basic items and giving conceptual uniformity to a model. Second, as different process models are often composed of common parts—e.g., the ones describing organization-wide procedures—mechanisms to abstract and reuse common characteristics across models are needed. The possibility of reusing existing models when developing new ones speeds up elicitation. Moreover, abstraction facilities that enable representation of common information in a single component help in keeping models consistent in later updates.

Consistency reconciliation. During elicitation, information about the process is collected piecemeal and many fragments that model different clusters of information are created. Inconsistencies and lack of relevant information are often discovered while this preliminary modeling is carried out, in particular during verification with the process user. Hence, the PML and the support tools must be flexible enough to tolerate inconsistencies and incompleteness, yet allowing the representation of meaningful models. This allows modelers to represent and verify the information they own, and later refine it into a consistent and complete representation.

1.2 E^3: Why Another Process Modeling Language?

E^3 is an academic project that has its roots in object-oriented (OO) research and conveys the idea that object-orientation can be used successfully for modeling software processes.

Our first experience in modeling a process with OO techniques used the Coad and Yourdon [Coad and Yourdon 1991a; Coad and Yourdon 1991b] OO analysis and design methods and supporting tools to model the process of a department of the FIAT car manufacturer [Baldi and Jaccheri 1995]. In order to provide a simulation tool for our experiments, we also developed a prototype PSEE based on distributed OO programming techniques [Baldi and Dall'Anese 1994].

This preliminary experience in using OO design and analysis techniques used on an *as-is* basis for process modeling demonstrated not only that "software processes are software too" [Osterweil 1987], but also that software processes are designs and requirements specifications, too. In other words, it is possible to model a software process at a high abstraction level by using pure OO analysis techniques without delving into low-level details. Moreover, our experience demonstrated also the effectiveness of object-orientation in eliciting process models, since the models were used also as a means to communicate information to the process users. By contrast, the experience with the PSEE, despite its prototypical nature, was resisted in the organization management that hardly perceived it as a real asset and considered it inapplicable on an organization-wide scale.

Based on this experience, we focused our research on the elicitation of process models through object-orientation. Nevertheless, despite the encouraging results, OO design and analysis techniques used on an *as-is* basis revealed some problems with respect to the requirements stated in Section 1.1:

—Syntax and semantics are not defined formally, thus preventing automatic analysis or simulation.

—Process-dedicated syntax constructs are needed in order to enhance process understanding. Although our experience showed that the techniques we employed increase process understanding, they also indicated that non-trivial process models can consist of hundreds of classes and associations that appear to the user as a flat web of identical boxes and arrows. Hence, in order to enhance understanding, process specific constructs mapped on the process components are needed, still using a graphical notation.

—Many OO tools [Larman 1998; Reenskaug et al. 1996; Rational Software Corp. 1997] cope with the complexity of OO models by employing view mechanisms. As these mechanisms are designed to provide general support for an OO model, and are focused on the OO modeling constructs, they do not consider the process modeling domain. Again, specific view mechanisms designed for software process model visualization are needed to help the user to inspect a model from different process perspectives.

—A widely recognized asset of the OO approach is its capability to support structure, reuse, and evolution. The features of a class can be reused and extended by means of inheritance. But what does it mean to reuse a class for which associations have been defined? What does it mean to change a class which participates in a set of associations with other classes? Again, process specific knowledge has to be added to implement evolution and reuse mechanisms.

Notably, the E^3 PML provides only rudimental support for attribute and method definition. For instance, it is not possible to specify the body of methods. In this respect, E^3 is less expressive than other PMLs, e.g., EPOS [Jaccheri and Conradi 1993]. However, this choice is in accordance with the methodologies proposed for OO analysis. As observed in [Embley et al. 1995], defining attributes and methods is concerned with design rather than analysis. Hence, our choice follows from our decision to build a system conceived specifically for capturing software process models rather than designing or enacting them.

1.3 Structure of the Paper

The paper is structured as follows. Section 2 provides a conceptual framework and essential terminology for the presentation of our system. In Section 3, the core of the paper, we provide a general overview of the E^3 language and a discussion about the role of associations in the general context of object-orientation, after which a detailed presentation of the language features follows. Section 4 describes the E^3p-draw tool that, in addition to providing support for the editing and archiving of E^3 models, embodies a view mechanism that can be regarded as an additional abstraction level besides the ones provided directly by the PML. Section 5 maps the features of E^3 against the requirements for elicitation stated in this section. The discussion is supported by reports on field case studies. Section 6 examines the relationships between E^3 and other projects that have addressed elicitation in the context of process modeling. Finally, Section 7 draws some conclusions about the E^3 project and highlights directions for future research.

2. E^3 TERMINOLOGY

Although there are good frameworks for software process modeling terminology [Lonchamp 1993; Feiler and Humphrey 1993; Conradi et al. 1992], none has yet been commonly accepted. In this work we start from the observation that, in a real world organization, the process rules to be followed during software production and the project's process that follows these rules coexist. No matter whether they are coded explicitly or followed implicitly, process rules usually have different scopes of applicability, ranging from the whole organization to a single project. They can be grouped according to the following kinds of real world processes (see Figure 1):

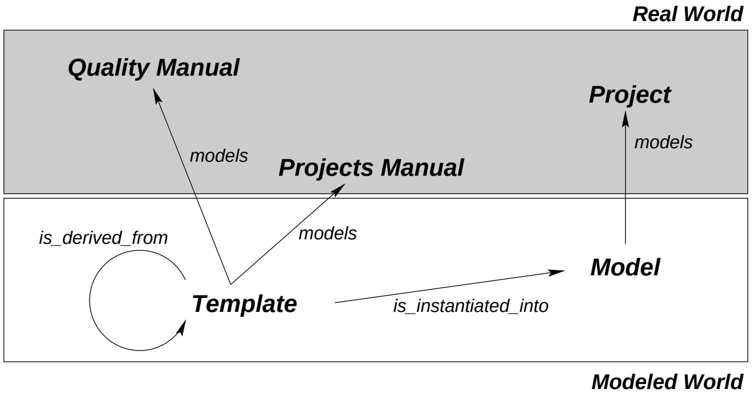

Fig. 1. Real world processes and process models.

Quality Manual. The set of rules for software production that must be adopted by the whole organization. These rules are usually described in an organizational manual. As an example, the rule "All the projects must include a development phase followed by a validation phase" might belong to a Quality Manual.

Projects Manual. The set of rules for software production that must be adopted in the context of a given set of projects. The rules of the Projects Manual must be compliant with those of the Quality Manual, although they can refine or extend the latter in order to specify the peculiarities of the given set of projects. Following the example above, the rule "All the projects must include a development phase exploiting object-oriented software construction" might belong to the Projects Manual describing projects that involve the development of telecommunication software.

Project. The set of real world entities involved in a given software production project. The Quality Manual and the Projects Manual are concerned with the process rules that are common to sets of Projects and, in order to maintain generality, do not usually describe facts like resource allocation and deadlines since they are likely to change from one project to another. In contrast, a Project contains tasks with actual deadlines, resources, products, rules, and tools. It must be compliant both with the Quality Manual and the Projects Manual. Returning to the example, the rule "The validation phase for the development of the GUI classes must be carried out by J. Doe, and it must end at most two days after the end of the development" might belong to a given Project in the area of telecommunication software.

The real world processes described so far can be represented by the following kinds of process models:

Template. Captures the key aspects of one or more Quality Manuals and Projects Manuals to describe the general issues which can be reused in the description of other similar processes, or to define a model which can provide guidance for a class of processes. In a Template there is no concern about the mapping of model entities onto projects. A Template can be refined into a new and more accurate Template. This is the case for a Projects Manual being described as a refined and extended version of the Template describing the corresponding Quality Manual.

Model. Captures the full details of a Project. Hence, it includes the mapping between the entities of a model and those of the real world. A Model is therefore concerned with allocation of resources and deadlines because they are essential information for the Project. A Model may be formal, thus providing the capability of performing "what if" analysis and simulation. As a particular case, a formal Model can also be enactable, providing machine support for the process performers involved in the Project. Each Model should comply to a Template.

3. THE E^3 PROCESS MODELING LANGUAGE

The E^3 PML has been inspired by work on object-orientation [Rumbaugh 1987; Coad and Yourdon 1991a; Coad and Yourdon 1991b; Rumbaugh et al. 1991; Booch 1991] to which it contributes by introducing and formalizing the management of associations. The language is given a formally defined syntax that enforces representation of well-formed models. Furthermore, it is given a graphical representation that facilitates the representation and communication of process models.

E^3 offers a set of built-in classes and associations conceived to describe the basic entities and relations involved in software processes. They can be customized by the modeler using inheritance. The language definition spans three conceptual levels, described in the remainder of this section: Creation, Definition, and Instance level. This three-tier structure of the language resembles the distinction among meta-classes, classes, and instances supported by some OO languages. However, it is a new contribution in the context of process elicitation and brings some advantages described in the following sections.

Besides the above separation in abstraction levels, modeling may take advantage of the powerful abstraction mechanisms of inheritance and aggregation in order to hide unnecessary or unknown detail. This is particularly useful during elicitation, where the information comes piecemeal and yet the modeler needs to communicate meaningful models to the process user in order to validate the information already gathered and identify those still missing. Using inheritance, the modeler is able to create classes and associations at a high level of abstraction without preventing a later refinement by specialization. Similarly, using aggregation, the modeler is able to hide the structure of the classes, and defer their detailed representation to the moment when the necessary information becomes available. The same mechanisms are also used as a structuring and reuse mechanism, together with instantiation. Inheritance provides a mechanism both for reusing of process elements and for structuring the knowledge about them. Aggregation defines how process elements are actually composed of other process elements.

Some of the mechanisms mentioned above have been extended in order to encompass the association concept that is central in our language and whose rationale is discussed in the next section.

3.1 Extending Object-Orientation with Associations

In object-orientation, a *class* provides a common description for a set of objects in terms of structure and behavior through attribute and method declaration, while an *association* describes a set of links among objects with common structure and semantics. Associations have been widely used by the database community [Chen 1976] as a modeling construct. An association declaration consists of a name, a sequence of related classes and a cardinality. As observed by Rumbaugh [Rumbaugh 1987], associations are a useful concept and need to be integrated in OO methods, as they facilitate the expression of declarative models, and help to keep evolving models consistent. In this vision, classes are a means to express knowledge about local structure and behavior, while associations express how classes can be related in building the global system.

The OO community has already addressed association management. OO analysis and design methodologies [Booch 1991; Coad and Yourdon 1991a; Rumbaugh et al. 1991] and the related informal notations introduce both built-in associations such as aggregation, use, or creation, and user-definable ones; however, they do not provide a formal characterization of associations. Furthermore, none of the most popular OO languages (e.g., C++, CLOS, and Smalltalk) devote special constructs to associations, which have to be implemented by references.

Our modeling approach aims at providing a means to represent associations among classes, since the structure and semantics of links among objects are as important as objects themselves. Process models must represent not only the entities involved in the process but also the relationships that relate and constrain such entities. The process modeling community has already identified [Conradi et al. 1992] a set of basic entities typical of every software process which can be modeled naturally in an object-oriented approach by using classes. Nevertheless, it is our opinion that it is also possible to identify a set of basic relationships existing among entities in a process, that can be modeled using associations. In addition, it is often possible to identify patterns of entities and relationships that occur in several process descriptions, and it is highly desirable to reuse these patterns whenever possible. In this context, associations are the language abstraction that allow us to "glue" entities in clusters that can be reused in different process models. The above considerations are supported by the observation and study of real software processes and related models.

In the literature on process modeling, different strategies to represent relationships between process entities are reported. For example, they can be represented explicitly, e.g., with the arcs that connect transitions to places in a Petri net-based process model, or implicitly, e.g., with a chain of pre- and post-conditions in a rule-based process model.

In the context of process modeling, we state the following requirements to integrate traditional OO concepts like objects, classes, and inheritance, with links and associations:

(1) Associations must be first-class elements, like classes. Hence, they must be organized in an inheritance hierarchy as well.

(2) If a class participates in an association, this knowledge should be inherited by its subclasses. A subclass must have a means to re-define such inherited

knowledge. This is crucial to allow a class to reuse not only local knowledge of its superclass, but also the associations its super class participates in.

(3) Relationships that are common to *every* software process have to be represented in the associated model to facilitate model inspection, understanding, and evolution.

(4) The same association, e.g., precedence among activities, can appear several times in a Template and the knowledge about a given association must be expressed by a single reusable item that also conveys information about how an association is visualized.

3.2 The Working Example

In order to improve the focus of our presentation we present the features of the E^3 language using a working example. It consists of a set of sentences that can be regarded as *fragments* from a quality manual. Hereafter, fragments are referred to by their number.

(1) Each software system is developed in four phases. Software development phases, i.e. requirement analysis, design, and coding, are strictly sequential. A fourth phase monitors software development and runs in parallel with the others.

(2) Each software development phase consists of a production phase and a validation phase. If the validation phase does not approve the result of the production phase, the production phase has to be re-executed by considering the feedbacks provided by the validation phase.

(3) In the context of a metrics program:
   (a) for each document, the number of pages, status, and number of solved and unsolved problems must be recorded;
   (b) for each phase, the start and finish times, number of hours worked, and number of iterations must be recorded.

(4) The input to each code validation phase is either an already tested document together with its test report or a document to be tested.

(5) A responsible person is identified for each design phase. Moreover, each design phase produces a design document with the support of a design tool.

(6) If the design phase must be carried out with an OO methodology:
   (a) those responsible for the phase must have been trained according to an OO training program;
   (b) the design document must adhere to a standard OO format and must be produced by using an OO design tool;
   (c) after the production phase, the responsible person decides non-deterministically to validate the design either by outsourcing the validation phase to an external consultant or by performing it internally.

3.3 E^3 Structure

The E^3 PML is structured into the following three conceptual levels:

Creation level. Where classes and associations are created by specializing those built-in.

Definition level. Where the elements created at Creation level are assembled to build a network of classes connected by associations. This network is actually the description of a Template.

Instance level. Where classes and associations belonging to a Template at the Definition level are instantiated into objects and links, which actually represent the entities belonging to a Model.

This paper focuses on the features of E^3 that are conceived for the representation of Templates. These are mainly related to the Creation and Definition levels. However, it will sometimes be necessary to explain the above features using concepts borrowed from the Instance level, since the three levels are conceptually related.

The relationship between Creation and Definition levels resembles the one between meta-classes and classes in OO languages such as Smalltalk. A Smalltalk program is defined as a set of classes that can be instantiated into objects. A meta-class is the class for a set of classes and its only purpose is to factor out characteristics, attributes and methods, that are common to a set of classes. Similarly, a created association is similar to a Smalltalk meta-class, i.e. it is not part of an E^3 Template and its only purpose is to factor out characteristics that are common to a set of association definitions. The common characteristics are the association name and the constraints on the kind of classes that can be related by the association definitions. In E^3, the Creation level (that can be called *meta-level*) has been conceived only for associations and not for classes as class level attributes and method representation is not a goal.

The *Creation level* includes both kernel[5] and user-defined classes and associations. Figures 2 and 3 show as an example the Creation level for a Template describing our working example. The *kernel* is unchangeable and constitutes the upper portion of the inheritance hierarchy of every Template. The kernel is organized in three roots. Kernel classes inherit from `E3Object` and kernel associations inherit from either `binary` or `ternary` associations[6]. *User-created* classes and associations customize the kernel for a given Template at the Creation level. For instance, in Figure 2 class `DesTool` is created by inheritance from kernel class `Tool`.

At the *Definition* level, classes are connected by means of associations to yield the *Class Network (CN)*, that is the actual representation of the Template. Figure 4 shows the CN representing the E^3 Template for our working example.

An E^3 Template is composed of both the inheritance tree built at the Creation level (Figures 2-3), which specifies *which* building blocks can be used in a CN, and the CN itself (Figure 4), which specifies *how* building blocks are related.

Reuse of common characteristics is achieved in two ways:

—information declared at Creation level, for classes (associations) namely, methods, attributes (plus arity and cardinality) are inherited by subclasses (subassociations);

[5]Kernel classes and associations are hereafter collectively referred to as the *kernel*.

[6]E^3 does not support n-ary associations with $n > 3$. Based on our experience with this version of the PML, we believe that ternary associations are sufficient for the software process modeling domain.

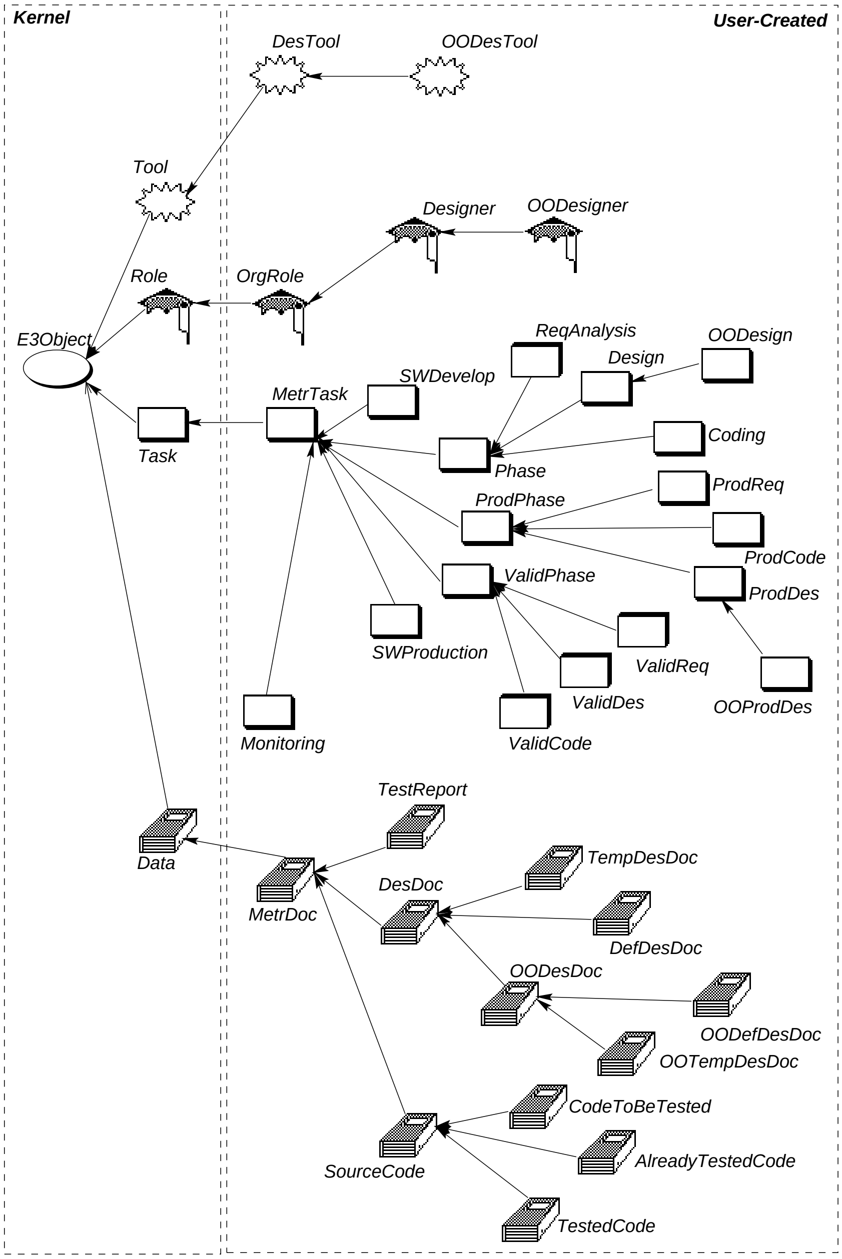

Fig. 2. E^3 PML Creation level classes (Class inheritance tree).

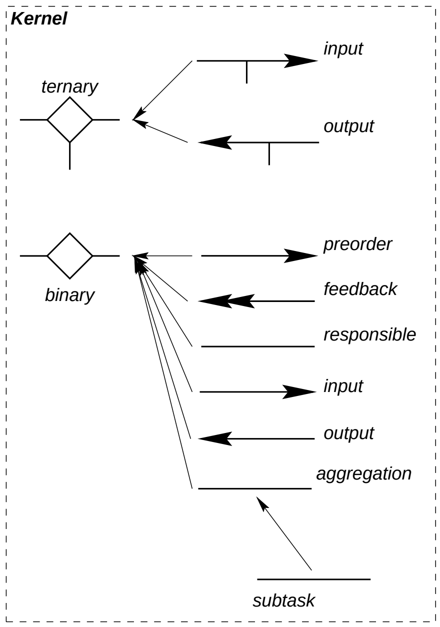

Fig. 3. E^3 PML Creation level associations (Association inheritance tree).

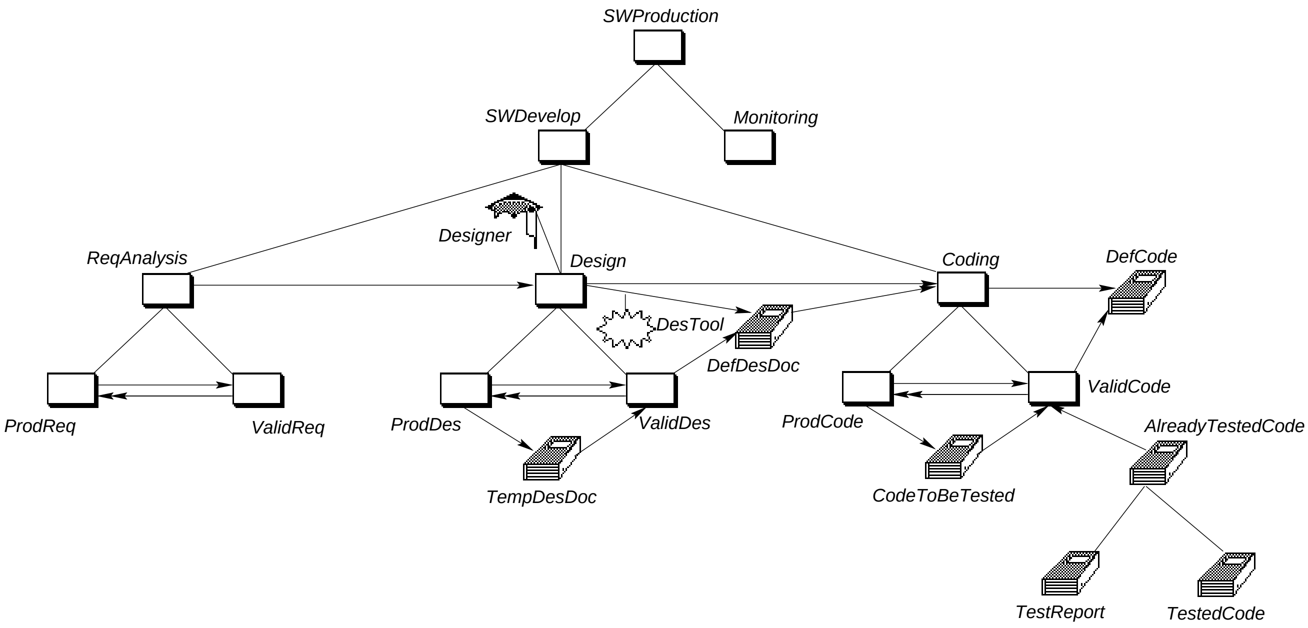

Fig. 4. E^3 PML Definition level (Class Network).

—associations defined for a given class C are inherited by those classes that are created as subclasses of class C at the Creation level. A mechanism to re-define inherited features is also provided.

Reuse leverages off of our association management mechanism, according to the observation made in Section 3.1 that different parts of the same Template often share global interconnection patterns rather than local information encapsulated in classes. Looking at our example in Section 3.2, Fragment 3 requires a description of documents and phases which can be described within a single class. On the other hand, the other fragments declare how building blocks are to be assembled together and how these interaction constraints must be shared by different parts of the quality manual. For example, Fragment 2 declares the structure of each development phase together with the constraints on the interaction among its components.

At *Instance* level, objects and links are created according to the constraints defined at Definition level. A class may have many instances and an association definition may have many links. A link l is a legal instance of an association definition A if the objects connected by l at the Instance level are instances of classes related by A at the Definition level.

This three-tier structure encompassing associations is original in the process modeling elicitation domain, although there exist PMLs conceived for enaction that use a multi-level schema [Warboys 1989] or even a multi-level schema supporting associations [Jaccheri and Conradi 1993]. Our schema allows a clear separation among abstraction levels, distinguishing among the creation of building blocks, their interconnection to form a Template, and the subsequent transformation in a Model. The key idea is the distinction between the Template that describes higher-level process information such as Quality and Projects Manuals, and the Model, that is a representation of the Project. This makes clear to modelers the abstraction level where they are working, *keeping distinct the goals of providing a reusable description and yet a fully detailed representation.*

Associations play a key role in this structure. As we stated in Section 3.1, we believe that they are fundamental in OO modeling and consequently we give a clear semantics to their use. Furthermore, we allow re-definition of associations, in order to facilitate reuse and extensibility of Templates. In addition, links (the Instance level image of associations) can be established to put additional constraints that are valid only for that particular model. Hence, a major feature in E^3 is the flexibility provided to modelers for deciding the right level of abstraction where an association needs to be defined and, possibly, re-defined.

3.4 Creation Level

The Creation level includes the kernel, described in detail in the next section, as well as user-created classes and associations. The operation provided by this level is *creation* through inheritance. Every class is created as a subclass of another class, from which it inherits attributes and methods. Every association is created as a subassociation of another association from which it inherits, in addition to attributes and methods, the classes participating in the association. An extra constraint is defined on associations: this and the others presented in the following are inspired by theoretical developments on object-orientation [Cardelli 1984].

An association $A'(\mathit{seqClasses}' : \mathit{seq}[\mathit{Class}])$ can be created by inheritance from an existing association $A(\mathit{seqClasses} : \mathit{seq}[\mathit{Class}])$ if[7]:

(1) Arity is preserved. Formally:

$$\sharp \mathit{seqClasses}' = \sharp \mathit{seqClasses}$$

(2) Each class in the sequence of classes related by A' is *compatible* with the respective class in the sequence of classes related by A. We say that a class C' is compatible with another class C, if C' belongs to the reflexive transitive closure[8] of relation inheritance of C. Formally, the following predicate must hold:

$$\forall\, i \in \operatorname{dom} \mathit{seqClasses}' \bullet \mathit{seqClasses}'\ i \in \mathit{inheritance}^{*}\, \mathit{seqClasses}\ i$$

3.5 Kernel Classes and Associations

Kernel classes are organized in an inheritance hierarchy rooted at `E3Object`:

—`E3Object` is the top-level superclass of each class.

—`Task` subclasses define common characteristics for a set of software process activities. Each `Task` instance denotes a process activity that can terminate either with or without approval. A `Task` instance can start when its predecessors have finished and its resources (input and roles) are ready. Each `Task` decides internally whether it terminates with or without approval. The notion of approval offered by the PML can be used to model that a verification and validation task can either approve the result of the respective production task, or require another iteration. For example, Fragment 2 of our example describes a general validation phase that can either approve the result of its respective production phase or require its re-execution.

—`Role` subclasses define responsibilities and represent a set of skills for a person. `Role` instances model actual resources with those responsibilities and skills. Fragment 6a introduces the role of a designer that must have been trained according to an OO training program. This is modeled by the `Role` subclass `OODesigner`, as shown in Figure 2.

—`Data` subclasses define software process artifacts or product components, like source code and anomaly forms. For example, Fragment 3a describes a class of documents that is modeled by the `Data` subclass `MetrDoc` as shown in the inheritance tree. Figure 5 provides the representation for classes `MetrDoc` and `MetrTask`, as requested by Fragment 3.

—`Tool` subclasses define methods of work. A `Tool` instance may represent a precise version of an automated tool or a written procedure. For example, Fragment 5 introduces a requirement about tools that is modeled by `Tool` subclass `DesTool`, as shown in Figure 2.

Kernel binary associations are as follows:

---

[7] $\mathit{seq}[X]$ is a sequence of elements of type X, modeled as a function from N to X, as in [Woodcock and Loomes 1988].

[8] f^* denotes the transitive reflexive closure of relation f, whereas f^+ denotes the transitive closure of relation f.

```
numPages:
status:
numProblemsFixed:
numProblemsNotFixed:
```

```
startTime:
finishTime:
numWorkedHours:
numIterations:
```

Fig. 5. Declaration of class MetrDoc and MetrTask.

—binary(E3Object′, E3Object″) is the top-level superclass of each binary association defined between two E3Object classes;

—aggregation(E3Object′, E3Object″) denotes composition. Definitions of association aggregation denote class composition and aggregation links denote object composition. Association aggregation provides one of the main abstraction facilities to represent how an entity can be refined into the composition of other entities. Figure 8 shows an example of class aggregation where AlreadyTestedCode is composed of TestReport and TestedCode.

—subtask(Task′, Task″) inherits from aggregation and denotes composition among Task subclasses. Association subtask offers one of the main organization dimensions of a process model. Indeed, activity decomposition is a well recognized way to structure work division in a production process. Both Fragments 1 and 2 denote examples of activity breakdown. Figure 4 shows the representation for Fragment 1 in which SWProduction is decomposed in two Task classes, SWDevelop and Monitoring. SWDevelop is further decomposed into ReqAnalysis, Design, and Coding.

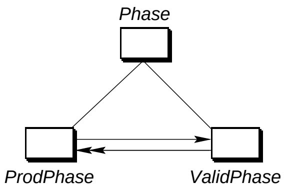

Fig. 6. The structure of class Phase at Definition Level.

—preorder(Task′, Task″) declares constraints on interactions among Task instances. If there exists a link[9] preorder$_l$(Task_o1, Task_o2), it means that Task_o2 can start only after Task_o1 has finished with approval. This is similar to a finish–to–start relationship in project management. Looking at Fragment 1, analysis, design, and coding are modeled by using three tasks connected by preorder association definitions, as shown in Figure 4.

—feedback(Task′, Task″) declares constraints both on instantiation and interaction of Task instances. We introduced the concept of feedback as a means to model activity loops that may occur in a Template, that is, to allow the modeler to describe the repeated execution of task patterns. For example, consider the

[9]An association at Creation level is denoted by $A(seqClasses : seq[Class]])$. An association definition occurrence, at Definition level, is denoted by $A_O(seqDefClasses : seq[Class])$. A link, at the Instance level, is denoted by $A_l(seqObject : seq[Object])$.

Template for Fragment 2 in Figure 6. The fragment states that if the validation phase does not approve the result of the production phase, the latter one has to be re-executed by considering the feedbacks provided by the validation phase. Hence, at the Instance level we have as many instances of `ProdPhase` and `ValidPhase` as many execution of the validation phase are required. A link `feedback`$_I$`(Task_o1, Task_o2)` denotes that if `Task_o1` finishes without approval `Task_o2` must start. In other words, a `feedback` link corresponds to a finish–to–start relationship in the case where `Task_o1` finishes without approval. Else, if `Task_o1` finishes with approval `Task_o2` does not need to start. This corresponds, in project management, to a task that has been scheduled and that does not need to be executed. Other PMLs model this kind of constraint by the same construct devoted to express precedence constraints [Jaccheri and Conradi 1993]. Nevertheless, we believe that these two relations have different semantics and consequently need to be represented with different abstractions.

—`responsible(Task, Role)` declares constraints on the interaction between resources and activities. An association definition `responsible`$_O$`(Task1, Role1)` denotes that the set of responsibilities declared for class `Role1` are needed for carrying out `Task1`. For example, the constraint of Fragment 5 is modeled by the association definition `responsible`$_O$`(Design, Designer)` (Figure 4). A link `responsible`$_I$ `(Task_o1, Role_o1)` denotes that human resource `Role_o1` is in charge of activity `Task_o1`.

—`output(Task, Data)` and `input(Task, Data)` describe information about the documents that are produced and consumed by `Task` classes. An association definition `output`$_O$`(Task1, Data1)` denotes that `Task1` instances produce as output `Data1` instances, and analogously for `input`. Figure 4 provides examples of `input` and `output` definitions, e.g., `output`$_O$`(Coding, DefCode)`.

Kernel ternary associations are as follows:

—`ternary(E3Object′, E3Object″, E3Object‴)` is the top-level superclass of each ternary association defined among three `E3Object` classes;

—`output(Task, Data, Tool)`. An association definition `output`$_O$`(Task1, Data1, Tool1)` denotes that `Task1` instances produce as output `Data1` instances by exploiting `Tool1` instances. As an example, the `output` association definition in Figure 4 provides a model for the second portion of Fragment 5 that states that each design phase (modeled as `Design`) produces a design document (`DefDesDoc`) with the support of a design tool (`DesTool`).

—`input(Task, Data, Tool)`. An association definition `input`$_O$`(Task1, Data1, Tool1)` constrains `Task1` instances to take as output `Data1` instances by exploiting `Tool1` instances.

3.6 Definition Level

At the Definition level, the modeler can build Templates by using the classes and associations created at the Creation level to produce a CN. The *CN origin* is the root of the decomposition hierarchy of a Template. The CN origin is the only `Task` subclass in a Template which is not a component of another `Task` aggregate according to an association `subtask` definition. The operations provided by this level are:

—*definition* of associations that have already been created at the Creation level;
—*re-definition* of associations that have already been defined at the Definition level.

The same association can occur more than once in a CN, i.e. the same association can have more than one association definition. On the other hand, a class is characterized by its attributes and methods, and by the association definitions it participates in. Thus, two occurrences of the "same class" (in terms of attributes and methods) that would occur in two different places of the CN cannot be represented as the same class, rather they are modeled by exploiting the subclass mechanism. If one wants to declare two occurrences of class C one has to declare two sub classes of class C identical to it except for the name, i.e. without declaration of attributes and methods.

For example, in Figure 4 class `DefDesDoc` and `TempDesDoc` have the same attributes and methods as `DesDoc`, as they specialize it, but they participate in different association definitions. Also, they model the existence of two different classes of instances. The first class is `TempDesDoc`, produced by `ProdReq` and taken as input by `ValidReq`, while the second is `DefDesDoc` produced by `ValidDes`, that also constitutes the output of the global design phase `Design`.

There are of course other solutions to represent Templates of classes and associations. We chose to duplicate a class each time it participates in different association definitions as in our approach the structure of the Template (or Model) is always regarded to be at least as important as the content of each class (or instance).

3.6.1 *Association Definition.* The occurrence of an association $A_O(seqDefClasses : seq[Class])$ can be *defined* from an association $A(seqClasses : seq[Class])$ already created if:

(1) Arity is preserved. Formally:

$$\sharp seqDefClasses = \sharp seqClasses$$

(2) Each class in the sequence of classes related by A_O is compatible with the respective class in the sequence of classes related by A. Formally:

$$\forall\, i \in \mathrm{dom}\ seqDefClasses \bullet seqDefClasses\ i \in inheritance^{*} seqClasses\ i$$

Semantics of association definition correspond to the usage of classes and associations created at the Creation level.

3.6.2 *Association Re-definition.* A class inherits all association definitions from its superclass. As an example, note that `ReqAnalysis`, `Design`, and `Coding` respect the constraint declared in Fragment 2 by inheriting `Phase` structure (see Figure 6). An inherited association definition $A_O(seqDefClasses : seq[Class])$ can be re-defined by $A'_O(seqDefClasses' : seq[Class])$ if:

(1) arity is preserved. Formally:

$$\sharp seqDefClasses' = \sharp seqDefClasses$$

(2) each class C' in the sequence of classes related by A'_O belongs to the non-reflexive transitive closure of relation inheritance of the corresponding class C in the sequence of classes related by A_O. Formally:

$$\forall\, i \in \mathrm{dom}\ seqDefClasses' \bullet seqDefClasses'\ i \in inheritance^{+} seqDefClasses\ i$$

We defined an inheritance mechanism for definition and re-definition as well. A class C' inherits from its superclass C:

—all the attributes and methods of C (as introduced before), and

—the associations already defined or re-defined for C.

Note that, if we would allow multiple occurrences of the same class C, a definition for an occurrence of C could not be inherited by a subclass of C because it would inherit all the definitions defined for the different occurrences.

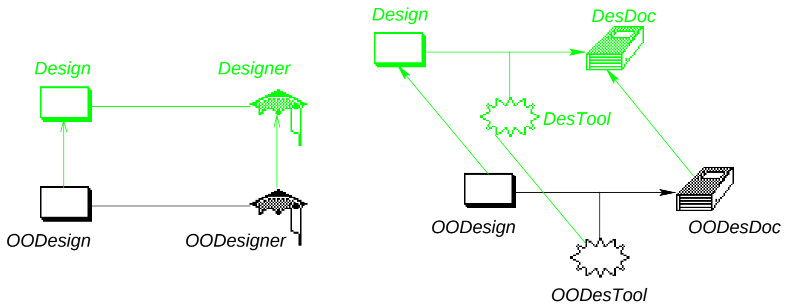

Fig. 7. Examples of association re-definition for binary and ternary associations.

The left hand side of Figure 7 shows an example of binary association re-definition. `OODesign` (and `OODesigner` as well) inherits from its super class `Design` all associations that have been defined for it, including `responsible`$_O$`(Design, Designer)`. In addition, we want to model the situation proposed in Fragment 6a that states that if the design phase is to be carried out with an OO methodology, those responsible for performing the phase must have been trained according to an OO training program. `OODesign` is a subclass of `Design` and `OODesigner` is a subclass of `Designer`: compatibility constraints are satisfied and `responsible`$_O$`(Design, Designer)` can be re-defined by `responsible`$_O$`(OODesign, OODesigner)`. Analogously, if one wants to specify that `OODesign` produces an `OODesDoc` by using an `OODesTool`, the ternary association `output`$_O$`(Design, DesDoc, DesTool)` has to be re-defined as in the right hand side of Figure 7. This latter re-definition models Fragment 6b.

3.7 Instance Level

A thorough definition of the Instance level is still the subject of on-going research. Nevertheless, this level aims at providing the modeler with the abstractions needed to derive a Model from the higher level Template already described at the Creation and Definition levels.

This is done by dealing with objects and links, that are instances of classes and association definitions, respectively. Associations constrain association definitions, whereas association definitions constrain links. As an example, association `subtask`, created between two `Task` classes, constrains each `subtask` definition to take place between two `Task` subclasses. Furthermore, if an association definition, e.g., `subtask`$_O$, is defined between `SWDevelop` and `Design`, this constrains each link instance of $\mathtt{subtask}_O$ to relate an instance of `SWDevelop` and an instance of `Design`.

3.8 Disjunction and Conjunction

Both a Template and a Model need a means to represent parallelism, non-determinism, and multiple alternatives. Traditional OO notations [Rumbaugh et al. 1991] devote special linguistic constructs, e.g., aggregation, to model composition of objects. Moreover, they usually delegate the representation of parallelism and non-determinism to complementary representations, e.g., StateCharts [Harel 1987]. Multiple alternatives can be expressed by means of cardinality constraints. E^3 uses disjunction and conjunction to express the above.

Disjunction is expressed by multiple definitions of the same association sharing a class, while conjunction is expressed by the definition of `aggregation` associations. This section discusses their meaning at the Definition and Instance levels. Informally, disjunction at Definition level allows representation of two or more alternatives one of which is chosen at instantiation time to be reflected in the Model. Conjunction at Definition level allows representation of two or more constraints, to be reflected in the Model. Disjunction at Instance level is the way to represent two alternatives, one of which will be chosen during the actual enaction of the Model in a Project. Conjunction at Instance level is the way to represent that one object consists of the composition of two or more different objects.

3.8.1 *Definition Level*

Disjunction. If an association A, other than `aggregation` and its subassociations, is defined more than once, say $A_O(C_i, C)$ and $A_O(C_j, C)$, with $C_i \neq C_j$, it means that each C instance is connected *non-deterministically* by a link either to a C_i instance **or** to a C_j instance. Figure 8 models Fragment 4 of the working example. Association `input` is defined between `ValidCode` and both `CodeToBeTested`, and `AlreadyTestedCode`. Thus, a `ValidCode` instance takes as input either a `CodeToBeTested` **or** an `AlreadyTestedCode` instance and not both. The actual choice depends on instantiation time parameters such as resource availability.

Conjunction. If `aggregation` or one of its subassociations is defined more than once, say $\mathtt{aggregation}_O(C, C_i)$ and $\mathtt{aggregation}_O(C, C_j)$, with $C_i \neq C_j$, it means that each C instance is connected both to a C_i **and** a C_j instance. In the Template fragment of Figure 8, class `AlreadyTestedCode` consists of the aggregation of `TestReport` **and** `TestedCode`. Thus, each instance of `AlreadyTestedCode` is connected both to a `TestReport` **and** a `TestedCode` instance.

3.8.2 *Instance Level*

Disjunction. If an association definition A_O, other than `aggregation` and its subassociations, is instantiated more than once, say $A_I(o_i, o)$ and $A_I(o_j, o)$, with $o_i \neq o_j$, it means that instance o interacts *non-deterministically* with o_i **or** o_j. This definition, when applied to `preorder` and `feedback` links, models non-deterministic execution of tasks. For example, in the situation depicted[10] in the right hand side

[10] Elements at the Instance level are represented using the same symbol of the corresponding elements at Definition and Creation levels. In the E^3p-draw tool, however, elements at different

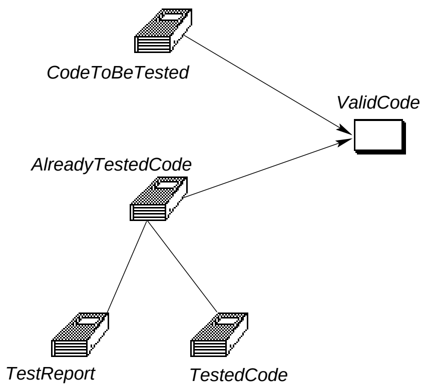

Fig. 8. An example of conjunction and disjunction at Definition level.

of Figure 9 `ValidDes_o1` and `ValidDes_o2` are two instances of class `ValidDes` with different attributes values denoting whether the phase is internal or external. In this situation, one of the instances will be chosen non-deterministically for execution at enaction time, as required by Fragment 6c.

Conjunction. If an `aggregation` link or one of its subassociations is instantiated more than once, say $\mathtt{aggregation}_l(o, o_i)$ and $\mathtt{aggregation}_l(o, o_j)$, with $o_i \neq o_j$, it means that instance o is decomposed into instances o_i and o_j. For association `subtask`, this is similar to a work breakdown relationship in the context of project management. The left hand side of Figure 9 shows an example. `SWProduction_o1` is decomposed into `SWDevelop_o1` and `Monitoring_o1`. Hence, *SWProduction_o1* does not terminate until both `SWDevelop_o1` and `Monitoring_o1` terminate. Since no `preorder` link is defined between them, the two sibling tasks are carried out in parallel.

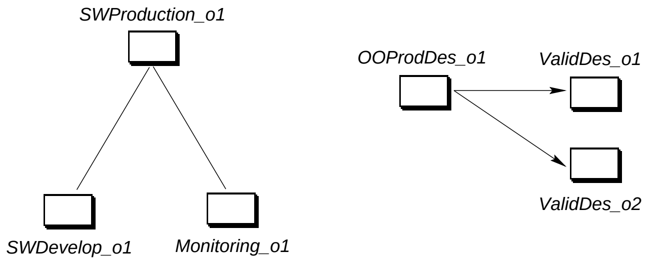

Fig. 9. Examples of conjunction and disjunction at Instance level.

4. E^3P-DRAW

E^3p-draw[11] can be regarded as an "upper" CASE [Fuggetta 1993] for software process modeling. It supports the development, analysis, inspection, and reuse of

levels are represented using different colors.

[11]Process Draw.

process models written using the E^3 PML. Model representation follows the three-layer distinction of the PML, thus enabling model editing, visualization, reuse, and analysis at Creation, Definition, and Instance levels. A view mechanism provides the user with the ability to customize the way a model is inspected. The unit of work and persistence of E^3p-draw is the *module*, which may contain process elements belonging to any of the three abstraction levels. Static analysis of process models is supported by a view mechanism and a simple query mechanism which enables property checking. Extension of this mechanism, as well as the provision of dynamic analysis and simulation tools integrated with E^3p-draw, is the subject of on-going work. Figure 10 shows the E^3p-draw *console*. It provides menus and

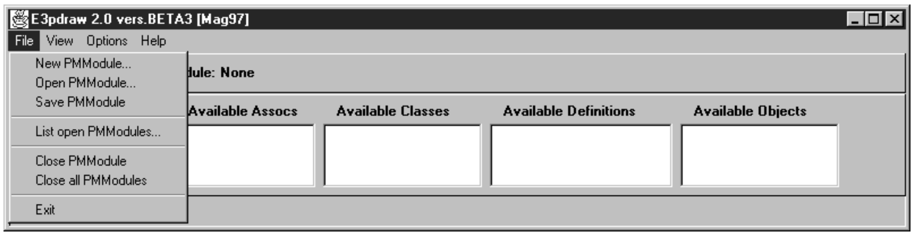

Fig. 10. E^3p-draw console.

buttons to invoke operations, and uses listboxes to display all the process elements—kernel and user classes and associations, together with their instances—currently available for model editing.

In the following sections, we briefly describe the features of E^3p-draw 2.0, available at [E^3 Team 1997]. In particular, Section 4.4 discusses design and implementation issues, in relation to earlier versions of the tool.

4.1 Modules

A *module* is a collection of process model elements possibly spanning all the three levels of abstraction provided by E^3. Conceptually, the module provides the encapsulation unit for all the elements needed in order to describe a process model, and can be considered the unit of work in E^3p-draw sessions. The module is also naturally the unit of persistence managed by E^3p-draw: modules can be saved to disk and later restored in a different session. Loading a module populates the console with the corresponding process elements. Many modules may be active at a time, although only one of them is the working module. The others are opened as read only, and may serve to provide availability of process elements to be reused in the working module.

Reuse is constrained by the module concept as well. In principle, reuse should enable the modeler to include an element in a working model that is already defined in the context of another model. Nevertheless, the meaning of a user-defined process element, as opposed to kernel elements, may be dependent on the context of the model where the element was placed. This "meaning" is represented in E^3 by the three abstraction levels. If a process element is imported from a module, the corresponding elements belonging to higher abstraction levels need to be imported as well. For instance, if an object is imported from another module, its class at

Creation and Definition levels must be imported in that module as well. It is worth noting that a single process element may constitute a relevant portion of a model, because of the use of aggregation.

4.2 Views

An E^3p-draw view displays a subset of the items belonging to a module. The set of elements visualized can be customized by the modeler, although a number of predefined views already exist.

E^3p-draw provides the so-called *base*, *inheritance*, and *derived* views. Base views are conceived for model editing, while derived views are conceived for model inspection. Inheritance views are in between, since they provide inspection of the model as far as the inheritance relationship is concerned, and yet they are editable.

Base views are used to actually create or modify a model. A base view is always bound to a given abstraction level. Using the buttons on the toolbar, the modeler can create, delete, or modify process elements belonging to the view's abstraction level. Attributes and methods can be defined on process elements. Figure 11 shows a base view at Definition level containing a portion of our example process. Syntax checks are always performed during model editing in a base view.

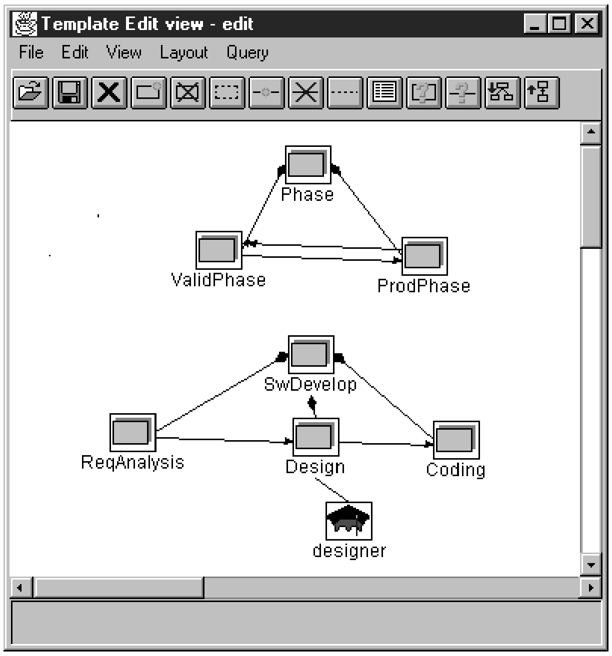

Fig. 11. E^3p-draw: a base view at Definition level.

Inheritance views visualize the inheritance tree, either at the Creation or Definition levels. A predefined inheritance view visualizes the kernel tree. In contrast, user-definable inheritance views are generated starting from a class selected by the user. The *superinheritance view* displays recursively the superclasses of the class selected, while the *subinheritance view* displays recursively the subclasses. For example, the inheritance relationships existing between `phase` and its subclasses `ReqAnalysis`, `Design`, and `Coding`, is visualized by the subinheritance view in Figure 12. In an inheritance view, the user can delete or modify classes belonging to the tree being visualized, create new classes, or invoke the visualization of a derived view for a given class.

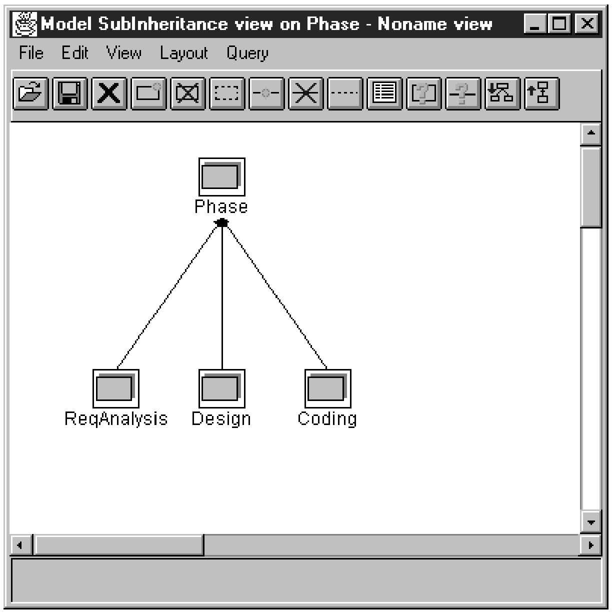

Fig. 12. E^3p-draw: an inheritance view at Definition level.

Derived views enable model inspection. Four kinds of derived views are provided in E^3p-draw: *simple*, *simple recursive*, *composite*, and *composite recursive* views. Basically, simple views visualize associations among classes, while composite views visualize aggregates together with the associations defined within them.

A simple view is defined for a class, and visualizes the class and all the association definitions the class participates in, except for `aggregation` associations. Hence, simple views make sense only at Definition and Instance levels. Figure 13 shows the simple view of class `Design`. The simple recursive view applies the simple view to a class, and then recursively to each class connected to it by an association definition other than `aggregation`.

The composite view is defined for a class, which is visualized together with all the definitions of `aggregation` (or its subassociations such as `subtask`) where the class participates as source class. Furthermore, the simple view of each class belonging to the aggregate is visualized as well. The composite recursive view shows the composite view of a class and then applies recursively the composite view on each class belonging to it. Figure 14 shows the composite view on class `Design`. Note that association definitions `subtask(Design, ProdPhase)` and `subtask(Design,`

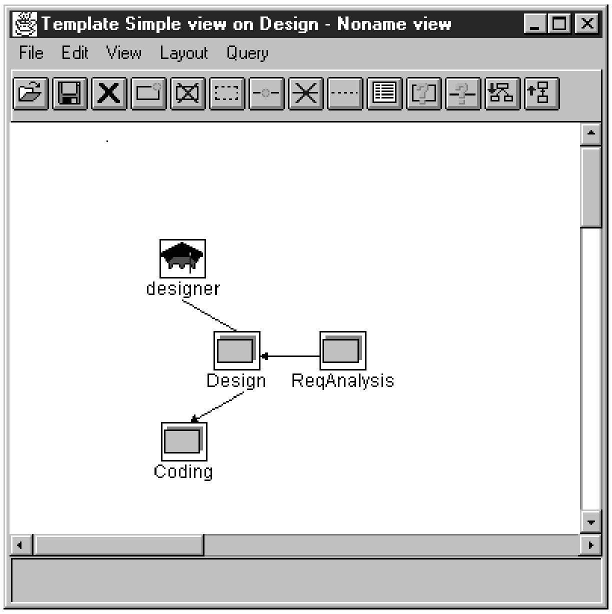

Fig. 13. E^3p-draw: a simple view at Definition level.

`ValidPhase`) are inherited, thus drawn in a different color and decorated with the label "`in`".

Derived views can be customized by hiding associations and classes. The user can specify which kind of associations need to be hidden, and whether or not to visualize nodes connected by a currently invisible association. This mechanism serves two purposes. On one hand, it may help the modeler during editing of particularly tangled models, by hiding temporarily unneeded associations. On the other hand, it can be used to define a "projection" for a given view of a set of associations. For example, for a given class, one may be interested in seeing only definitions of associations `input` and `output` to consider data flow, or only definitions of associations `preorder` to look at control flow.

Finally, base and derived views at the Definition level provide an automated instantiation feature. The invocation of this operation generates a new base view at the Instance level containing an instance for every process element contained in the base view at the Defined level. Additional instances can be defined by the user. Figure 15 shows a base view at Instance level which was obtained by automated instantiation of the composite view of the class `Design`.

Each view can be saved on persistent storage, either to save incomplete work or for later retrieval as an aid during model inspection. In the current implementation, the modifications made to the underlying model after a view has been saved are not applied automatically when the view is restored. A user-driven mechanism is being developed to cope with this issue.

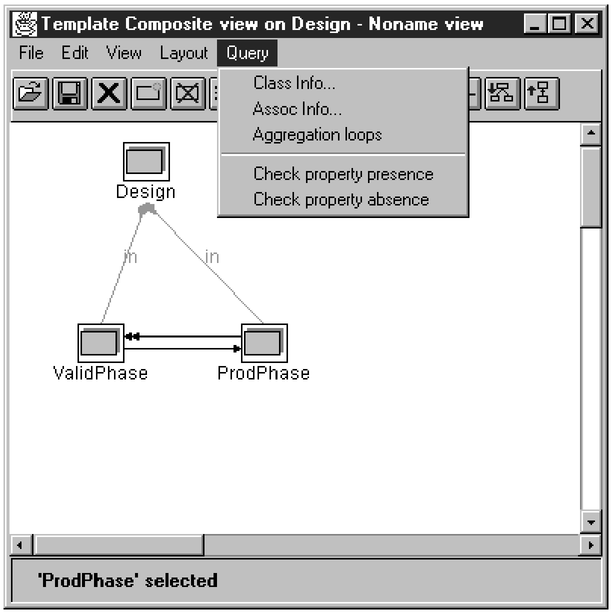

Fig. 14. E^3p-draw: a composite view at Definition level.

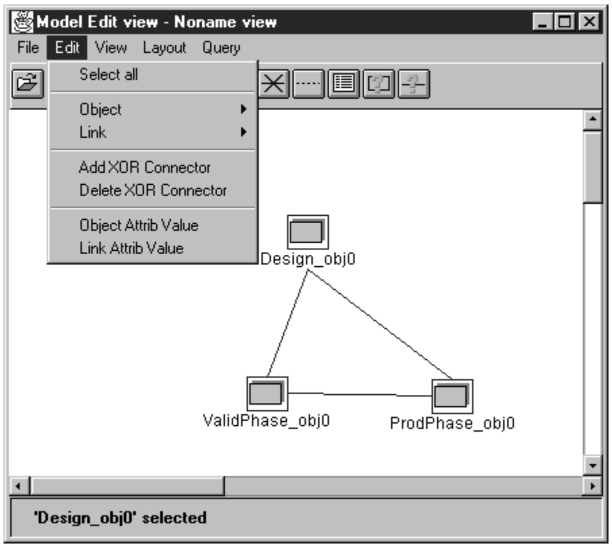

Fig. 15. E^3p-draw: a base view at Instance level.

4.3 Queries

E^3p-draw provides a query mechanism that is designed to support static analysis of process models. At the time of writing, the query mechanism supports only elementary checks on the topology of the model, as determined by the definition of associations. For instance, it is possible to detect the presence of loops in an `aggregation` tree. More generally, the tool provides support for checking whether a given property of association definition holds or not. For instance, one can check whether all the tasks of a given module have a `responsible` definition or, similarly, can evidence the tasks that lack a `responsible` definition. A query can be performed in the context of a whole module, or of a view.

4.4 Design and Implementation Choices

The initial release of the tool, E^3p-draw 1.0, supported only the Creation and Definition levels, along with views designed to allow inspection of a model according to the information, functional, and structural perspectives. These views were the *Inheritance* view, that retained the same meaning in version 2.0, the *Task* view, focused on the relations a given task participates in, and the *Task Synchronization* view, focused on the decomposition of a given task through the `subtask` relation and on the `preorder` and `feedback` relationships. As discussed in the next section, we used E^3p-draw 1.0 extensively for modeling real world processes. The experience revealed the need for more powerful and flexible mechanisms, described earlier and currently implemented in version 2.0. Such mechanisms can be used to derive the three perspectives above, if needed. In fact, the Task view is a specialization of a simple view, while the Task Synchronization view can be obtained by a composite view applied to a Task where all the associations but `preorder` and `feedback` are hidden.

Another issue was support for multi-user editing. E^3p-draw 1.0 exploited a client-server architecture where the server managed the centralized model repository. Nonetheless, in our experience, providing multi-user support introduced complexity in design and implementation without substantial advantages—in many organizations, even if the characteristics of the process model are determined by a team, its actual development is often carried out by a single person. For this reason, E^3p-draw 2.0 has been implemented as a single-user application.

A related issue was represented by portability. Version 1.0 was developed in C++ under DEC Ultrix because of its reliance on the object-oriented DBMS Object/DB we used as a model repository, and the Interviews library was used for GUI programming. Nevertheless, it became clear that support for PC boxes was highly desirable, due to their increasing pervasiveness. The implementation of version 2.0 minimized portability concerns through the adoption of the Java language [Sun Microsystems 1994]. Java enabled portability on all platforms supported, which presently include PCs as well as Unix boxes, and also provided a uniform API for GUI programming. Moreover, the use of a "true" object-oriented language opened up interesting developments as far as simulation is concerned. Currently, E^3p-draw elements are mapped onto Java classes and objects. E^3 presently leaves the behavior of methods unspecified. Specifying such behavior with the Java language would lead to a nice integration of E^3p-draw with subsystems providing dynamic analysis, simulation, or even enaction.

As for the repository, at the time of writing the modules are stored as text files written in a pre-defined syntax. We made this choice in order to be able to distribute our system without depending on an underlying database.

5. DISCUSSION

The E^3 PML and the E^3p-draw tool are conceived to provide support for process model elicitation through abstractions and mechanisms that are easily understandable by process users and yet defined formally. The following discusses how the characteristics offered by the E^3 system fulfill the requirements presented in Section 1.1. However, we start by giving a background of field case studies which provided feedback and inspiration for the design of our system.

5.1 Experiences with E^3

For each of these case studies, the corresponding E^3 model can be retrieved at [E^3 Team 1997].

5.1.1 *FIAT/Iveco.* This first case study [Baldi and Jaccheri 1995] was aimed at modeling the quality manual of one of the software departments of the company—the problem was initially approached by using plain OO methods. The meta-process we used when performing the elicitation with E^3 was fairly simple and characterized by a low degree of interaction with process users in the organization. First, we studied the documentation provided by the process users, then developed a Template of the process manual, and finally presented it to process users for discussion. The presentation consisted of an explanation of the characteristics of the E^3 language and a discussion of the Template. The presentation of E^3 was tailored for the audience, which included people without knowledge of computer science, and focused on the concepts needed to understand a model, rather than on the concepts needed by the modeler. In retrospective, the impact of this case study on the organization was fairly poor: the management was never really involved in the initiative.

5.1.2 *Olivetti.* Aimed at modeling the quality manual of the Research&Development PC&Server department in Olivetti [Milano 1996], this second case study benefited from the lessons learned during the first one, and by a stronger commitment of management and process users in the initiative. This allowed us to accurately plan the elicitation phase. Elicitation consisted of a study of the documentation, development of a preliminary Template, and 25 meetings during which the Template fragments were used as a whiteboard to achieve a common comprehension of the process and foster discussion among the process modeler and the process users responsible for different parts of the manual. During and after each meeting, the fragments were gradually integrated by help of association definitions. A set of problems, mainly inconsistencies, in the original process description was found, that were recognized by process users. This triggered an update of the quality manual, which now actually includes E^3 Template fragments as a precious aid to understanding.

5.1.3 *Experience with graduate students.* The E^3 system has been used for four years in a software engineering course being taught at Politecnico di Torino to describe the software process prescribed for all the project activities. This case study [Jaccheri and Lago 1996; Jaccheri and Lago 1997] differs from the previous ones for two reasons. First, process users are computer science students with a strong background in modeling and programming. Second, the Template was developed to meet educational requirements and not elicited from a real process. Therefore, understandability and consistency reconciliation cannot be evaluated on the basis of this case study. On the other hand, the main result of this case study is the full exploitation of the three conceptual levels to achieve the reuse of the same Template over the years.

5.2 Features and Requirements

The features offered by the E^3 system are: object-orientation, associations, explicit distinction among levels of abstraction, kernel classes and associations, disjunction and conjunction, together with the view mechanisms provided by the tool. The following analyzes these features in relation to the requirements for elicitation stated previously. In doing this, we support our statements by borrowing experience from the case studies described earlier. The discussion is summarized in Table 1. When applicable, the table cells show the degree of support provided by a characteristic of the E^3 system in fulfilling a requirement posed by elicitation.

	Understanding and domain coverage	*Structure and reuse*	*Consistency reconciliation*
Object-orientation	Secondary	Primary	Secondary
Associations	Secondary	Secondary	Primary
Explicit abstraction levels	Secondary	Primary	
Kernel classes and associations	Primary	Primary	
Disjunction and conjunction	Primary		
Customizable views	Primary	Secondary	Secondary

Table 1. E^3 features and elicitation requirements.

5.2.1 *Model understanding and domain coverage.* Kernel classes and associations are the primary support for achieving understanding. As demonstrated by our first two case studies, process users feel familiar with the basic concepts of E^3 after a short explanation. Also, to understand a Template fragment, it is sufficient to be acquainted with the informal meaning of kernel classes and associations because of the intuitive graphical notation. Concerning domain coverage, our case studies show that kernel associations are sufficient most of the times. However, some new associations were created for both the first and the second case studies to model the product structure.

Our object-oriented framework extended by associations enables the modeling of the different perspectives of the process in a uniform way. Our solution differs from classical ones. For example, Object Modeling Technology [Rumbaugh et al. 1991] prescribes that the system structure must be modeled by classes and associations, and delegates the modeling of system functions and control to other formalisms (Data Flow Diagrams and StateCharts [Harel 1987]). An underlying problem of this approach is the difficulty in maintaining consistency in the different models, and increasing complexity for the user.

The view mechanism of the tool is crucial to present Template fragments in such a way that the process user can understand the process by observing it from different and yet coherent perspectives.

Disjunction and Conjunction have been used thoroughly in our case studies mainly to model task parallelism and non–determinism and, together with product aggregation, are present in each of the software processes of our case studies.

5.2.2 *Model structure and reuse.* Kernel classes and associations are the building blocks that are used in each E^3 Template. As discussed in Section 3.5, kernel elements factor out knowledge that is typical for each software process and was repeated several times in the quality manuals we examined during the first two case studies.

Object-orientation is the primary support for structure and reuse. Local information about each class can be reused by inheritance in the context of the same Template. As an example, the classes `MetrDoc` and `MetrTask` mentioned in Section 3.5 were the superclasses of each document and activity class in the Template of our third case study. Moreover, patterns of knowledge can be reused by inheritance thanks to our mechanism that combines association inheritance and re-definition. This mechanism has been used in each of the three case studies. For example, during the first case study we developed the fragment displayed in Figure 6 that models a general phase structure to be reused by several Template activities, and we reused it by specialization throughout the Template.

The issues related to the three conceptual levels were investigated mainly in the context of the third case study. As each year a different project problem is proposed for the course by an external customer, we use the Template to derive instantiated models that, for the time being, we represent as Microsoft Project plans. The mapping from the Template to an instantiated model was done manually, as the E^3 system does not yet implement these facilities. We learned that a Template can be used as a trace to instantiate models and we gained insights about how to design mechanisms to support instantiation within the next version of our system. Implementation of these features is in its prototype phase, hence we are just starting evaluation of the effectiveness of our system in that respect. However, on the basis of our initial experience, we claim that the project manager (or whichever person in charge of creating project plans) is able to reuse all the information defined in a Template, e.g., activity names and descriptions, activity structures, and activity precedence constraints, while building an actual plan for the model.

5.2.3 *Consistency reconciliation.* Inheritance and aggregation are the language features that allow the modeler to develop incomplete model fragments and then specialize them gradually. Our field experience shows that aggregation is used extensively during elicitation, because classes are decomposed into aggregate classes as knowledge on the process increases. This initially hides incomplete information without limiting understandability. In turn, inheritance is mostly used when the final Template is realized. During model integration inheritance is exploited to generalize common patterns in super classes that are then specialized in several places of a same Template, by adding information.

Associations are used during integration as a means of binding together different Template fragments built during early phases of elicitation. This is possible as associations are first order elements of the language and they are not represented as part of classes.

As experienced during the second case study, incomplete Template fragments can be produced and visualized effectively by the help of the view mechanism provided by the tool. The different fragments can be produced gradually and the problem of ensuring total consistency among the different fragments can be delayed. During this activity, the query mechanism provided by the tool, although still elementary, can be used to perform static analysis.

6. RELATED WORK

Elicitation of information is not confined to the research area of software process modeling, rather it spans several areas in computer science. A great deal of research on elicitation belongs to the research area concerning of requirements engineering [Zave and Jackson 1997] which, as far as software engineering is concerned, focuses on the requirements for a software product. Elicitation of information is also central in building an expert system. This task involves elicitation of knowledge about the domain, often directly from an expert. Here, methodologies, techniques, and tools exist for this task [Darimont et al. 1997; Jackson 1990; Zave and Jackson 1997]. In both cases, however, representation of the information collected is only part of the problem. Since information is often extracted directly from an expert, inter-person interaction plays a key role in the elicitation activity. Process model elicitation is similar in this respect, although in this paper we only addressed the problem of providing a suitable support for the representation activity, rather than considering a comprehensive methodology for process model elicitation.

Furthermore, process model elicitation is not concerned necessarily with software process models, rather it is gaining attention in the more general context of *Business Process Reengineering* (BPR). The term BPR was coined by Hammer to denote the "fundamental rethinking and radical redesign of business processes" [Hammer 1990] to achieve improvement. A model of the business process is central in the redesign activity, although emphasis is usually on building a model to capture the requirements of the new process being redesigned rather than the properties of the old one being enacted. There is increasing awareness of opportunities for synergies among researchers in the areas of software process modeling and BPR [Warboys 1994; Huff 1993], and some approaches have already been proposed which try to integrate and leverage off of the results in the two domains [Graw and Gruhn 1995].

We now concentrate on software process modeling, where a number of field case studies [Kellner and Hansen 1989; McGowan and Bohner 1993; Aumaitre et al. 1994; Bandinelli et al. 1995; Barghouti et al. 1995] provided evidence of the problems concerned with process model elicitation. Existing systems have different means to address the issue of supporting the elicitation activity. We briefly summarize the characteristics of systems providing support for elicitation and compare these characteristics with those provided by the E^3 system.

6.1 "Non-PM" notations

Notations and tools not explicitly developed for process modeling have been used without any adaptation for eliciting processes.

Madhavji et al. propose a framework for elicitation called Elicit [Madhavji et al. 1994] structured in three dimensions: View, Method, and Tool. The View dimension defines the set of process elements which are relevant for the visualization of a process, e.g., roles and activities, together with their properties. The Method dimension contains the definition of a meta-process for elicitation, where the Tool dimension is concerned with the tool support for the elicitation activity. This framework does not mandate a particular tool for elicitation, however they report about the use of Statemate and a tool called Elicit, developed in the same project. The two have been used in a complementary way: Statemate provides support for graphical representation and dynamic analysis, while Elicit is a textual interface that guides the user during the activity of collecting and structuring process information but does not provide any kind of model representation per se. In this context, the E^3 system could be used as a tool for elicitation.

Kellner et al. [Kellner and Hansen 1989] report about eliciting and specifying a process model that consists of three sub-models: a functional model represented by Activity Charts, a behavioral model represented by Statecharts, and a structural model, represented by Module Charts. The relations among these models are expressed by natural language documents. Heineman [Heineman 1994] reports about translating automatically StateCharts models into a model enactable with the Marvel system, mentioned later.

The E^3 system extends the Statemate approach in several directions. First, it provides a unified object-oriented framework that is easier to learn than multiple orthogonal modeling frameworks, where the perspective can be reconstructed through views. The view mechanism of E^3 extends the one presented by Kellner because it is based around a unified formally defined representation of the process and not on different unrelated models. Then, it offers formally defined associations to relate different model fragments, to better facilitate reuse and consistency among fragments. Further, the work in [Kellner and Hansen 1989] does not report any exploitation of a process modeling oriented kernel. Finally, E^3 provides multi-level process representation that distinguishes between Templates and instantiated models. On the other hand, for the time being, no experience in translating automatically E^3 models into enactable ones has yet been acquired.

McGowan et al. [McGowan and Bohner 1993] report about using IDEF0, a standardized version of a subset of SADT. No experience about the use of a support tool is reported. Here, object-orientation is not exploited, and there are no process modeling oriented kernel or view mechanisms. SADT does not distinguish between class and instance level, thus providing a single level representation of the process. Associations to model document input and output as well as task sequencing are provided. However, no support for new associations is given. Last, the meaning of multiply defined associations is not defined formally.

In general, the advantages of using existing notations are evident: know-how and support tools are often already available, hence skill can be reused and little training is needed. However, this approach suffers from some general drawbacks:

language constructs are poor in the context of the process modeling, since they are not process specific. Consequently, the resulting specification is somewhat complex and non intuitive. Also, no dedicated feature for process reuse is offered.

6.2 Existing PMLs

Existing PMLs, mainly conceived for implementation and enaction, are adapted and used for elicitation.

SLANG, the PML of the SPADE [Bandinelli et al. 1994] environment, can be classified in this category. SLANG is a formal language that exploits Petri nets for behavior definition, object-orientation for structure definition, and provides special features for process model evolution. Although conceived for enaction, [Bandinelli et al. 1995] reports a successful use of SLANG for process elicitation. For the purpose of elicitation, E^3 PML extends SLANG by providing a richer kernel. SLANG does not offer process modeling oriented building blocks, rather it delegates their creation to the modeler. Also, SPADE does not offer any view mechanism devoted to process modeling. Finally, although SLANG is based on Petri nets and the mapping from SLANG to Petri nets is specified formally, the integration between the Petri nets and the object-oriented component is not.

Process Weaver [Ett and Becker 1994] is based on an internal PML, hidden to the user, that can be manipulated through a number of views. Elicitation is facilitated by the highest abstraction level called Method Level where decomposition hierarchies of activity types can be defined. An activity type can be regarded as a specification, or interface, that can be associated with a textual description of related roles, products, and techniques. Process Weaver offers decomposition as its main abstraction facility. On the other hand, object-oriented inheritance is not provided. Also, some relationships are modeled informally as textual descriptions, thus hindering inconsistency discovery and subsequent reconciliation, as well as efficient model structuring.

The advantages of exploiting a PML aimed at enaction, come from the possibility of transforming gradually a model specification into an enactable implementation without linguistic transformations. However, it is difficult to separate the user oriented perspective from the enaction one. Also, these PMLs tend to concentrate on behavioral rather than functional or informational issues. For example, Process Weaver's Method Level concentrates only on activity aspects and not on tools, data, human roles, and associations among them.

A notable exception to the above is constituted by Endeavors [Bolcer and Taylor 1996] and its predecessor Teamware [Young and Taylor 1994] which have addressed the problem of model understandability since their conception. The Endeavors PML and system are designed for technical as well as non-technical users like managers, domain experts, accounting, marketing, and support staff. Models are expressed in an object-oriented fashion and provide the framework in which process models can be easily written, read, composed, and reused. Users can create new definitions, which can inherit the properties of existing ones. The system provides only predefined views to browse process models.

6.3 Extensions of already existing PMLs

Some PMLs and PSEEs have been extended by a layer providing support for elicitation.

Marvel 3.1 offers a process modeling language called MSL, with three main constructs being classes, rules, and tool envelopes. Classes have attributes that can be typed attributes or links to other classes.

Barghouti et al. [Barghouti et al. 1995] report on the modeling of two real world processes. For each of them, they first specified a model using Marvel's MSL and then generated a set of Marvel rules for it, thus obtaining the environment to enact the process scenarios. During enaction both timing and resource utilization information can be collected for further assessment and validation. The meta-processes they followed are different for the two case studies. One of the processes did not have a consistent description. Before presenting the MSL specification, the authors present an informal flow diagram that is similar to an E^3 Template, although not formal. The authors write that "…coding a model of a realistic process in MSL is cumbersome and requires more modeling sophistication than is present in many organizations. A higher-level graphical notation may be more suitable, especially during the early phases of process modeling and analysis…".

MSL and the E^3 PML have been developed with two different goals: MSL aims at providing models from which to derive automatically enactable process models, whereas the goal of the E^3 PML is to express models that can be understood by process users. This is a difficult problem, that can also be scaled up to the software specification domain. It is questionable whether specification languages that can be translated automatically into code are more or less important than notations that have the only goal of providing understandable models, e.g., DFDs.

More recently, the Activity Structured Language (ASL) [Kaiser et al. 1994] has been implemented in the context of Marvel. It is a formally defined language with the purpose of providing specific support for elicitation and specification. ASL is a variant of Riddle's constraint expressions that enables the modeling of process model topology by means of regular expressions extended with concurrency operators. ASL fragments can be translated into MSL ones.

OIKOS [Montangero and Ambriola 1994] offers two PMLs which are both concurrent logic languages. LIMBO is an elicitation language, while Paté is an enactment language. Like the E^3 PML, LIMBO offers high level process modeling oriented kernel concepts, e.g., office, role, coordinator. On the other hand, the relationships among these elements cannot be defined declaratively. Neither view mechanisms nor reuse facilities are currently provided.

IPSE 2.5 PML [Bruynooghe et al. 1994] has been extended with a language based on temporal logic, called Base Model (BM), to support process high level descriptions and analysis. BM syntax and semantics are formally defined, and enable property proofs. BM and E^3 are complementary rather than comparable: E^3 would benefit of formal definition of semantics as in BM and, on the other hand, E^3 is more suitable for process understanding, structure and reuse as it provides process modeling oriented graphic syntax, views, and kernel.

7. CONCLUSIONS

We believe that, despite the emphasis traditionally put on enaction, process modeling research can *already* help real world organizations in the elicitation and modeling of their software processes. In this work, we showed requirements specific to elicitation and described a PML that is conceived especially for elicitation of process models.

The separation in three levels of abstractions and the central role of formal management of associations in an object-oriented framework are the key features of E^3. Nevertheless, such formally specified concepts do not limit the understandability of the models developed with the language. Also, understandability is further improved by the graphical notation and the view mechanism provided by the E^3p-draw tool. The effectiveness of the approach has been shown by reporting real world case studies and by comparing the features of E^3 with those of other PMLs.

We are continuously improving and assessing E^3 in the context of the following research threads:

Formal semantics. We are defining a formal semantics that describes the dynamic aspects of our PML. The goal of this activity is to provide automated support for *analysis* of process models by enabling verification of dynamic properties, as well as to define precisely the aspects of the Instance level. The static aspects of the Instance level, described informally in this paper, are already described in [Jaccheri 1996a]. An initial contribution about dynamic aspects can be found in [Corno et al. 1996], where the semantics of the language is given by means of re-writing rules that translate E^3 PML expressions into a process algebra based on CCS [Milner 1989].

Reuse. Our association management mechanism allows different parts of the same Template to share a common structure by inheriting association definitions. This solves the problem of providing intra-Template reuse [Jaccheri 1996b]. Nevertheless, inter-Template reuse, i.e. reuse across different Templates, is still the subject of on-going work.

Better information hiding. Our field case studies revealed that better encapsulation and information hiding mechanisms are seldom needed. We are considering the definition of a scope mechanism for associations in which different visibility levels can be assigned to association definitions.

Integration with the Web. There is a trend in reorganizing information systems around intranets that use a Web browser as the unique interface to applications. We are gaining leverage off of the fact that E^3p-draw is written in Java, and investigating a Web-based infrastructure to provide access to the process models of an organization through a version of the tool integrated into a browser. This would simplify greatly the deployment of the application, and enable concurrent access of process models by several users into the organization, which could benefit the integration of informal, HTML-based descriptions of the process, with formal E^3 models that can be inspected by the user.

Validation with industrial partners. The industrial case studies we performed provided us with valuable feedback on the effectiveness of our design choices. Assessing our system from field experience is an activity which is always in parallel to the development of the system. However, existing software process modeling systems, including E^3, lack a validation based on a quantitative rather than qualitative assessment of the benefits. More work has to be done in this direction.

ACKNOWLEDGMENTS

We wish to thank Silvano Gai, Mario Baldi, Enrico Amerio, Paolo Senesi, Gian Franco Bottini, Tatiana Rizzante, Amato Milano, and Alessandro Bonaudo. Special thanks go to Alfonso Fuggetta for his many suggestions about early drafts of this paper.

REFERENCES

AUMAITRE, J., DOWSON, M., AND HARJANI, D. 1994. Lessons learned from formalizing and implementing a large Process Model. In *Proc. of the European Workshop on Software Process Technology* (Feb. 1994), pp. 227–240.

BALDI, M. AND DALL'ANESE, T. 1994. A distributed object-based system for process model enaction. In *Proc. of AICA'94 Italian Annual Conf.* (Sept. 1994).

BALDI, M. AND JACCHERI, M.L. 1995. An Exercise in Modeling a Real Software Process. In *Proc. of AICA'95 Italian Annual Conf.* (Sept. 1995).

BANDINELLI, S., FUGGETTA, A., GHEZZI, C., AND LAVAZZA, L. 1994. SPADE: An Environment for Software Process Analysis, Design, and Enactment. pp. 223–247. In [Finkelstein et al. 1994].

BANDINELLI, S., FUGGETTA, A., LAVAZZA, L., LOI, M., AND PICCO, G.P. 1995. Modeling and Improving an Industrial Software Process. *IEEE Trans. on Software Engineering 21*, 5 (May), 440–454.

BARGHOUTI, N., ROSENBLUM, D., AND BELANGER, D. 1995. Two Case Studies in Modeling Real, Corporate Processes. *Software Process: Improvement and Practice 1*, 1.

BASILI, V. AND ROMBACH, H. 1988. The TAME Project: Towards Improvement-Oriented Software Environments. *IEEE Trans. on Software Engineering 14*, 6 (June), 758–773.

BOLCER, G. AND TAYLOR, R. 1996. Endeavors: A Process System Integration Infrastructure. In *Proc. of the 4^{th} Int. Conf. on Software Process (ICSP4)* (Dec. 1996).

BOOCH, G. 1991. *Object Oriented Design with Application.* The Benjamin/Cummings Publishing Company Inc.

BOOTSTRAP PROJECT TEAM. 1993. Bootstrap: Europe's assessment method. *IEEE Software 10*, 3, 93–95.

BRUYNOOGHE, R. ET AL. 1994. PADM: Towards a Total Process Modelling System. pp. 293–334. In [Finkelstein et al. 1994].

CARDELLI, L. 1984. A Semantics of Multiple Inheritance. In G. KAHN, D. MACQUEEN, AND G. PLOTKIN Eds., *Semantics of data types*, Volume 173 of *LNCS*, pp. 51–67. Springer.

CHEN, P. 1976. The Entity-Relationship Model—Towards a Unified View of Data. *ACM Trans. on Database Systems 1*, 1 (March), 9–36.

COAD, P. AND YOURDON, E. 1991a. *Object-Oriented Analysis.* Prentice-Hall.

COAD, P. AND YOURDON, E. 1991b. *Object-Oriented Design.* Prentice-Hall.

CONRADI, R., FERNSTROM, C., FUGGETTA, A., AND SNOWDON, R. 1992. Towards a Reference Framework for Process Concepts. In *Proc. of the European Workshop on Software Process Technology (EWSPT'92)*, LNCS (Sept. 1992), pp. 3–18. Springer.

COOK, J. AND WOLF, A. 1994. Toward Metrics for Process Validation. In *Proc. 3^{rd} Int. Conf. on Software Process* (Oct. 1994). IEEE-CS Press.

CORNO, F., CUSINATO, M., JACCHERI, M.L., PRINETTO, P., AND RIZZANTE, T. 1996. Defining Formal Semantics for the E^3 Software Process Modeling Language. Technical Report 2/96, DAI, Politecnico di Torino, Italy.

CURTIS, B., KELLNER, M., AND OVER, J. 1992. Process modeling. *Comm. of the ACM 35*, 9 (Sept.), 75–90.

DARIMONT, R., DELOR, E., MASSONET, P., AND VAN LAMSWEERDE, A. 1997. GRAIL/KAOS: a Requirements Engineering Environment. In *Proc. of the 19th Int. Conf. on Software Engineering* (May 1997), pp. 612–613. ACM Press.

DEITERS, W. AND GRUN, V. 1994. The FUNSOFT Net Approach to Software Process Management. *Int. Journal on Software Engineering and Knowledge Engineering 4*, 2, 229–256.

DOWSON, M., NEJMEH, B., AND RIDDLE, W. 1991. Fundamental Software Process Concepts. In *Proc. of the 1st European Workshop on Process Modeling (EWPM'91)* (1991), pp. 15–37. Italian Society of Computer Science (AICA) Press.

E^3 TEAM. 1997. E^3 Web page. `http://www.idi.ntnu.no/~letizia/E3DIR/e3.html`.

EMBLEY, D., JACKSON, R., AND WOODFIELD, S. 1995. OO Systems Analysis: Is It or Isn't It? *IEEE Software*, 18–33.

ETT, W. AND BECKER, S. 1994. Evaluating the Effectiveness of Process Weaver as a Process Management Tool: A Case Study. In *Proc. of the 3rd Symp. on Assessment of Quality Software Development Tools* (June 1994), pp. 204–223.

FEILER, P. AND HUMPHREY, W. 1993. Software Process Development and Enactment: Concepts and Definitions. In *Proc. of the 2nd Int. Conf. on Software Process* (March 1993). IEEE-CS Press.

FINKELSTEIN, A., KRAMER, J., AND NUSEIBEH, B. Eds. 1994. *Software Process Modelling and Technology*. Research Studies Press/John Wiley & Sons.

FUGGETTA, A. 1993. A Classification of CASE Technology. *IEEE Computer 26*, 12 (Dec.), 25–38.

GHEZZI, C., JAZAYERI, M., AND MANDRIOLI, D. 1991. *Fundamentals of Software Engineering*. Prentice-Hall International Editions.

GRAW, G. AND GRUHN, V. 1995. Distributed modelling and distributed enaction of business processes. In *Proc. of the 5th European Software Engineering Conf.*, Volume 989 of *LNCS* (Sept. 1995), pp. 8–27. Springer.

HAMMER, M. 1990. Reengineering Work: Don't Automate, Obliterate! *Harvard Business Review*, 104–112.

HAREL, D. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming 8*, 3 (June).

HEINEMAN, G. 1994. Automatic translation of process modeling formalisms. In *1994 Centre for Advanced Studies Conf. (CASCON)* (November 1994), pp. 110–120.

HUFF, K. 1993. From software process engineering to business process reengineering. In *Proc. of the 8th Int. Software Process Workshop* (March 1993), pp. 98–101. IEEE Computer Society Press.

HUFF, K. 1996. Software Process Modeling. In A. FUGGETTA AND A. WOLF Eds., *Trends in Software: Software Process*, Chapter 1. John Wiley & Sons Ltd.

HUMPHREY, W. AND KELLNER, M. 1989. Software Process Modeling: Principles of Entity Process Models. In *Proc. of the 11th Int. Conf. on Software Engineering* (1989).

JACCHERI, M.L. 1996a. A Z Specification of the E^3 PML and tool. Technical Report 4/96, DAI, Politecnico di Torino, Italy.

JACCHERI, M.L. 1996b. Reusing Software Process Models in E^3. In *Proc. of the 10th Int. Software Process Workshop* (June 1996).

JACCHERI, M.L. AND CONRADI, R. 1993. Techniques for Process Model Evolution in EPOS. *IEEE Trans. on Software Engineering 19*, 12 (Dec.), 1145–1156.

JACCHERI, M.L. AND LAGO, P. 1996. Teaching Process Improvement in an Industry Oriented Course. In *Proc. of the British Computer Society Quality SIG Int. Conf. on Software Process Improvement—Research into Education & Training (INSPIRE'96)* (Sept. 1996).

JACCHERI, M.L. AND LAGO, P. 1997. Applying Software Process Modeling and Improvement in Academic Setting. In *Proc. of the 10th ACM/IEEE-CS Conf. on Software Engineering Education and Training* (April 1997).

JACKSON, P. 1990. *Introduction to Expert Systems*. Addison Wesley.

KAISER, G. AND FEILER, P. 1987. An Architecture for Intelligent Assistance in Software Development. In *Proc. of the 9th Int. Conf. on Software Engineering* (April 1987), pp.

180–188.

Kaiser, G., Popovich, S., and Ben-Shaul, I. 1994. A Bi-Level Language for Software Process Modeling. In *Configuration Management*. John Wiley & Sons.

Kellner, M. and Hansen, G. 1989. Software Process Modeling: A Case Study. In *Proc. of the 21st Annual Hawaii Int. Conf. on System Sciences* (Jan. 1989).

Krueger, W. 1992. Software Reuse. *ACM Computing Surveys 24*, 2, 131–183.

Larman, C. 1998. *Applying UML and Patterns—An Introduction to Object-Oriented Analysis and Design*. Prentice-Hall.

Lonchamp, J. 1993. A Structured Conceptual and Terminological Framework for Software Process Engineering. In *Proc. of the 2nd Int. Conf. on Software Process* (Feb. 1993).

Madhavji, N., Höltje, D., Hong, W., and Bruckhaus, T. 1994. Elicit: A Method for Eliciting Process Models. In *Proc. 3rd Int. Conf. on Software Process* (Oct. 1994).

McGowan, C. and Bohner, S. 1993. Model Based Process Assessment. In *Proc. of the 15th Int. Conf. on Software Engineering* (May 1993).

Mi, P. and Scacchi, W. 1990. A Knowledge-Based Environment for Modeling and Simulating Software Engineering Processes. *IEEE Trans. on Knowledge Engineering and Data Engineering 2*, 3 (Sept.), 283–294.

Milano, A. 1996. Il modello E^3 del manuale di qualità della divisione Olivetti Progetto PC&Servers. Master's thesis, Politecnico di Torino, Italy. In Italian.

Milner, A. 1989. *A Calculus of Communicating Systems*, Volume 92 of *LNCS*. Springer.

Montangero, C. and Ambriola, V. 1994. OIKOS: Constructing Process-Centred Environment. pp. 131–152. In [Finkelstein et al. 1994].

Osterweil, L. 1987. Software Processes are Software Too. In *Proc. of the 9th Int. Conf. on Software Engineering* (March 1987), pp. 2–13.

Paulk, M., Curtis, B., Chrissis, M., and Weber, C. 1993. Capability Maturity Model, Version 1.1. *IEEE Software*, 18–28.

Rational Software Corp. 1997. The Rational-Microsoft family of Visual Modeling Tools. Technical report, Rational Software Corp. White paper. Available at `http://www.rational.com/`.

Reenskaug, T., Wold, P., and Lehne, O. 1996. *Working With Objects—The OOram Software Engineering Method*. Manning.

Rumbaugh, J. 1987. Relations as Semantics Constructs in an Object-Oriented Language. In *Proc. of the ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)* (Oct. 1987), pp. 466–481. In ACM SIGPLAN Notices 22(12), Dec. 1987.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. 1991. *Object-Oriented Modeling and Design*. Prentice Hall.

Sun Microsystems. 1994. The Java Language: A White Paper. Technical report, Sun Microsystems. Available at `http://www.java.sun.com/`.

Tanaka, T., Sakamoto, K., Kusumoto, S., Matsumoto, K., and Kikuno, T. 1995. Improvement of Software Process by Process Visualization and Benefit Estimation. In *Proc. of the 17th Int. Conf. on Software Engineering* (April 1995), pp. 123–132.

Taylor, R. et al. 1988. Foundations for the Arcadia Environment Architecture. In *Proc. of the 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Environments* (Nov. 1988), pp. 1–13. In ACM SIGPLAN Notices 24(2), February 1989.

Warboys, B. 1989. The IPSE 2.5 Project: Process Modelling as the basis for a Support Environment. In *Proc. of the 1st Int. Conf. on System Development Environments and Factories* (May 1989). Pitman Publishing.

Warboys, B. 1994. Reflections on the relationship between BPR and Software Process Modelling. In *Proc. of the 13th Int. Conf. on the Entity-Relationship Approach*, Volume 881 of *LNCS* (Dec. 1994). Springer.

Woodcock, J. and Loomes, M. 1988. *Software Engineering Mathematics*. Pitman Publishing.

YOUNG, P. AND TAYLOR, R. 1994. Human-Executed Operations in the Teamware Process Programming System. In *Proc. of the 9th Int. Software Process Workshop* (Oct. 1994).

ZAVE, P. AND JACKSON, M. 1997. Four Dark Corners of Requirements Engineering. *ACM Trans. on Software Engineering and Methodology 6*, 1 (Jan.), 1–30.