Proc. 16th Int. Conf. Software Engineering
May 16-21, 1994, Sorrento, Italy. pp. 159-168.

Understanding "Why" in Software Process Modelling, Analysis, and Design

(Research Paper)

Eric S. K. Yu and John Mylopoulos
Department of Computer Science, University of Toronto
Toronto, Ontario, Canada M5S 1A4

Abstract

In trying to understand and redesign software processes, it is often necessary to have an understanding of the "whys" that underlie the "whats" – the motivations, intents, and rationales behind the activities and input-output flows. This paper presents a model which captures the intentional structure *of a software process and its embedding organization, in terms of dependency relationships among actors. Actors depend on each other for goals to be achieved, tasks to be performed, and resources to be furnished. The model is embedded in the conceptual modelling language Telos. We outline some analytical tools to be developed for the model, and illustrate how the model can help in the systematic design of software processes. The examples used are adaptations of the ISPW-6/7 benchmark example.*

Keywords: software process modelling, requirements engineering, organization modelling, actor dependency.

1 Introduction

A software process refers to the set of tools, methods, and practices used to produce a software product [11]. Historically, software development have largely been product-centered. Recently, many researchers and practitioners have refocused their efforts on the process dimension of software engineering (e.g.,[12, 13, 14]).

At the core of most of these efforts is some way of describing or modelling a software process. In this paper, we propose a model that aims to capture the motivations, intents, and rationales that underlie a software process.

Current process models have been proposed to address a variety of needs, e.g., to improve understanding, to facilitate communication or management, or to support and sometimes automate process enactment [3]. Most of these models aim to express *what* steps a process consists of, or *how* they are to be performed. However, in order to improve or redesign a process, we often need to have a deeper understanding about the process – an understanding that reveals the "whys" behind the "whats" and the "hows".

Typically, process performers need models that detail the "hows", process managers prefer models that highlight the "whats", while process engineers charged with improving and redesigning processes need models that explicitly deal with the "whys" [1]. The need for different types of software process models for different purposes may be compared to the need for different languages to represent software *products* at different levels – requirements (providing the "why"), design (specifying the "what"), and implementation (giving the "how") (e.g., [15]).

The need to capture design rationales behind software *products* is well recognized (e.g., [21]). However, to address *process* rationale, we need to face up to the distributed, organizational nature of processes. Because software processes are carried out by many parties or individuals, the "whys" for a process are typically not dictated by some process engineer, but reflect the complex social relationships among process participants. When considering different options for improvement, software engineers, managers and other stakeholders in the organization need to understand how each option would affect their daily work, and their pursuit of project and personal goals. This deeper understanding would help them choose process design options that meet their needs and interests.

We assume that process participants are organizational actors who need to cope with problems cooperatively on an on-going basis. How actors make use of, and constrain, each others' problem solving activity is therefore an important aspect of a software process that needs to be modelled and reasoned about. In the *Actor Dependency* model, actors depend on each other for *goals* to be achieved, *tasks* to be performed, and *resources* to be furnished. By modelling the structure of these *intentional dependencies* among actors, we provide a higher level characterization of a software

[1] We follow [17] in distinguishing these three classes of users of software process models.

process.

The model distinguishes among four types of dependencies, reflecting the types of freedom allowed by one actor on the other in a dependency relationship. *Commitment* and *criticality* characterize the strength of a dependency. Dependencies are threaded through *roles* and *positions*, as well as physical *agents*, creating an intricate web of relationships that we call the *intentional structure* of the software process. The model is embedded in the conceptual modelling language Telos [18]. The semantics of the modelling concepts are characterized using intentional concepts developed in agent modelling in AI (e.g., [2]).

The Actor Dependency model allows an analyst to explore opportunities open to actors by matching wants against abilities, to identify vulnerabilities of actors arising from their dependencies, and to recognize channels by which actors can mitigate their vulnerabilities, such as mechanisms for *enforcing* a commitment, *assuring* its success, and *insuring* against failure. The ability to assess these broader implications help differentiate among alternatives in efforts to design or redesign software processes.

The perspective on software process modelling, analysis, and design adopted in this paper is based on our work in requirements engineering. In requirements engineering, there has been recognition for some time that understanding and modelling the environment is an important part of systems development. More recently, frameworks have been proposed that treat system and environment as a whole to be jointly modelled, analyzed, and designed (e.g., [7, 4, 8, 6, 19]).

In our approach, we emphasize the need to model how actors deal with problems on an on-going basis, by modelling how they relate to each other at an intentional level. The basic Actor Dependency model has been proposed in the context of information systems requirements engineering [24]. This present paper applies the model to the software process domain, and extends our earlier results in several ways. It shows how the model can be embedded in a conceptual modelling framework, making use of structuring mechanisms such as classification and generalization. It outlines analytical tools for the model, and illustrates its use in a design context. It also elaborates on the concepts of roles, positions, and agents, using a richer example setting than in previous papers.

Section 2 of this paper presents the modelling concepts of the Actor Dependency model. Section 3 sketches the formal representation of the model. Section 4 outlines the types of analyses that can be supported by the model. Section 5 places the model in the context of process design. Section 6 discusses the proposed approach in relation to existing research. Section 7 draws some conclusions from our work and outlines future work.

2 An Actor Dependency model

The basic features of the Actor Dependency (AD) model have been presented in an earlier paper [24], and are briefly reviewed in section 2.1. The concepts are illustrated using a simple example of a software project organization. Section 2.2 extends the basic model by distinguishing roles and positions from agents. Dependencies across role/position/agent relationships reflect the more elaborate and subtle aspects of software processes. The example used is an adaptation of the ISPW 6/7 benchmark example [16].

2.1 The basic model

An Actor Dependency model consists of a set of nodes and links. Each node represents an **actor**, and each link between two actors indicates that one actor depends on the other for something in order that the former may attain some goal. We call the depending actor the **depender**, and the actor who is depended upon the **dependee**. The object around which the dependency relationship centres is called the **dependum**. By depending on another actor for a dependum, an actor (the depender) is *able* to achieve goals that it was not able to do without the dependency, or not as easily or as well. At the same time, the depender becomes *vulnerable*. If the dependee fails to deliver the dependum, the depender would be adversely affected in its ability to achieve its goals.

Figure 1 shows an Actor Dependency model for a hypothetical (and simplistic) software engineering project organization. A customer depends on a project manager to have a system developed. The project manager in turn depends on a designer, a programmer, and a tester to do the technical work and be on schedule. Technical team members depend on each other for intermediate work products such as the design, code, and test results. The manager is also depended on by his boss for no project overrun, and by the quality assurance manager for the system to be maintainable. The user depends on the project manager for a user-friendly and high performance system.

The Actor Dependency model distinguishes among three main types of dependencies, based on the ontological category of the dependum, namely, assertion, activity, or entity [9].

In a **Goal Dependency**, an actor depends on another to make a condition in the world come true. Because only an end state or outcome (expressed as an assertion about the world) is specified, the dependee is given the freedom to choose how to achieve it. In the example of Figure 1, the goal dependency relationships between the project manager and his staff means that it is up to members to decide how to do their job. The customer does not care how the system is developed. It is the outcome that matters.

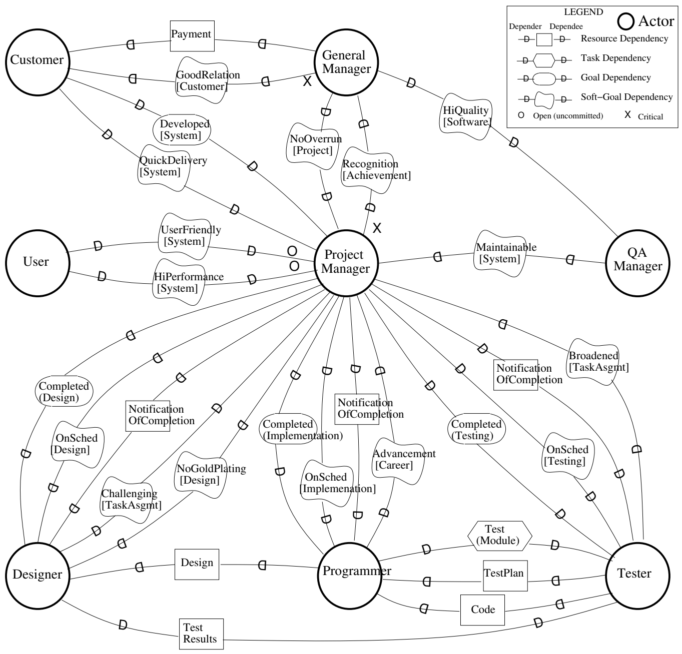

Figure 1: An Actor Dependency Model of A Simple Software Project Organization

In a **Task Dependency**, an actor depends on another to perform an activity. The depender's goal for having the activity performed is not given. The activity description specifies a particular course of action. The programmer depends on the tester to test a module via a task dependency by specifying a test plan. If the project manager were to indicate the technical steps for each team member to carry out, then the manager would be relating to his staff by task dependencies.

In a **Resource Dependency**, an actor depends on another for the availability of an entity (physical or informational). The depender takes the availability of the resource to be unproblematic (not requiring problem solving, from the depender's standpoint). In Figure 1, the general manager's dependency on the customer for payment, the tester's dependency on the programmer for code, and the project manager's dependency on his technical staff for notification of task completion, are modelled as resource dependencies.

A fourth type of dependency, **Soft-Goal Dependency**, is a variant of the first. It is different from a (hard) goal dependency in that there is no *a priori*, cut-and-dry criteria for what constitutes meeting the goal. The meaning of a soft-goal is specified in terms of the methods that are chosen in the course of pursuing the goal. The dependee contributes to the identification of alternatives, but the decision is taken by the depender. The notion of soft-goal allows the model to deal with many of the usually informal concepts. For example, the project manager's dependency on his boss for recognition can be achieved in many different ways. What constitutes sufficient recognition needs to be worked out between the two, and is ultimately decided by the depender. Treating "no project overrun" and "on schedule" as soft-goals indicates that these are not evaluated as binary yes/no assertions. A (hard) goal dependency would be used if there is a sharp cutoff, e.g., if the product must be delivered either by a promised date, or not delivered at all.

The four types of dependencies differentiate how the depender and dependee relate to each other in terms of their freedom in solving problems and achieving goals. We also distinguish among three degrees of strength.

In an **Open Dependency**, a depender would like to have the dependum goal achieved, task performed, or resource available, so that it could be used in some course of action. But failure to obtain the dependum would not affect the depender's goals to any great extent. On the dependee side, an open dependency is a claim by the dependee that it is able to achieve the dependum for some depender.

In a **Committed Dependency**, the depender has goals which would be significantly affected – in that some planned course of action would fail – if the dependum is not achieved. Because of its vulnerability, a committed depender would be concerned about the viability of the dependency. On the dependee side, a committed dependency means that the dependee will try its best to deliver the dependum, e.g., by making sure that its own dependencies are viable.

In a **Critical Dependency**, the depender has goals which would be seriously affected – in that all known courses of action would fail – if the dependum is not achieved. In this case, we assume that the depender would be concerned not only about the viability of this immediate dependency, but also about the viability of the dependee's dependencies, and the dependee's dependee's dependencies, and so forth. In Figure 1, the general manager's critical dependency on having good relations with the customer would lead him to be concerned about whether the project manager can (and will) develop a system and deliver it quickly to the customer, and whether technical team members will also do their part.

2.2 Roles, positions, agents, and associations

The basic Actor Dependency model can be extended by refining the notion of actor into notions of role, position, and agent.

A **role** is an abstract actor. Dependencies are associated with a role when these dependencies apply regardless of who **plays** the role. For example, the role "Monitoring Project Progress" depends on progress reports from team members, regardless of who is doing the monitoring.

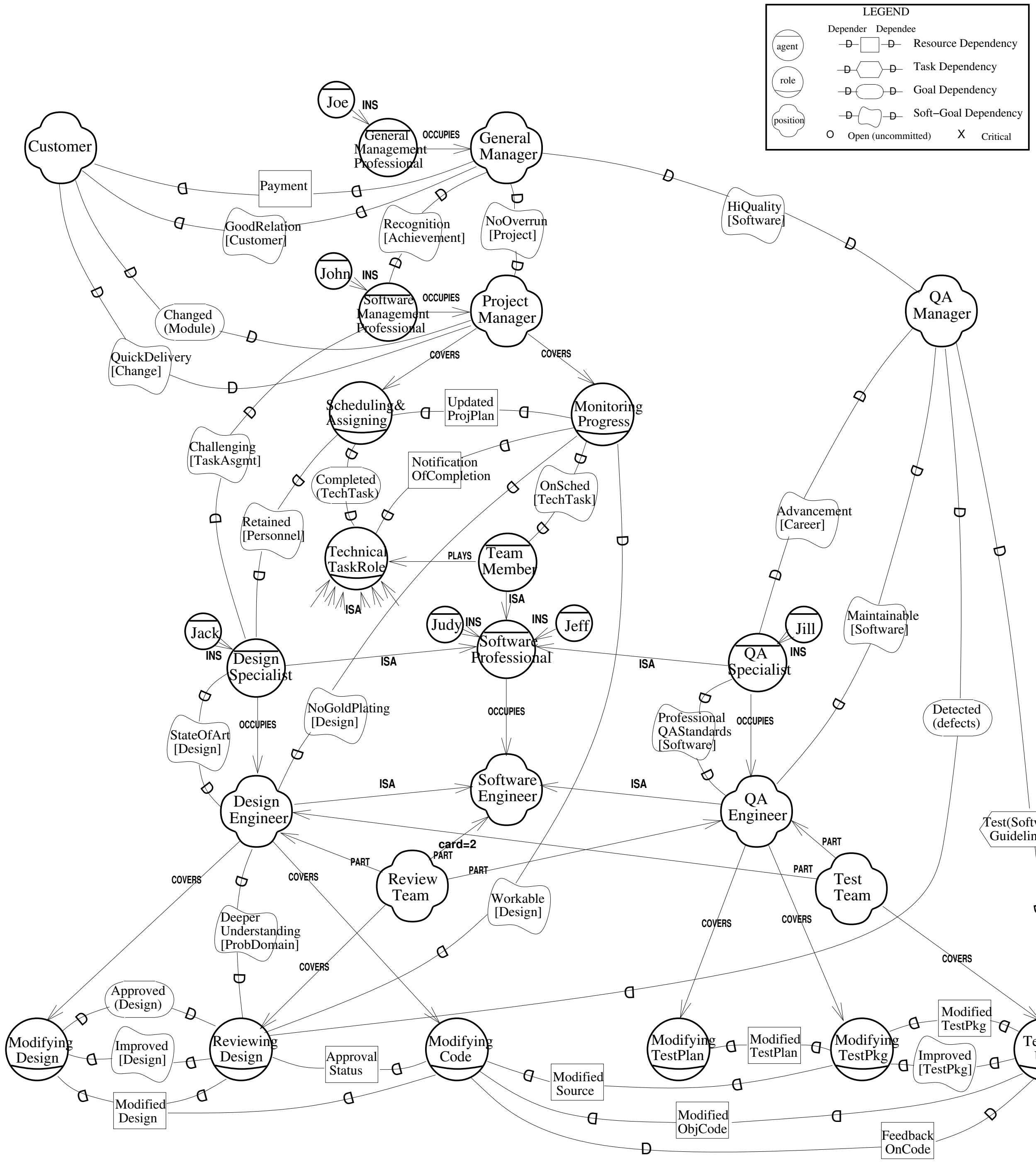

Figure 2: An Actor Dependency model with roles, positions, and agents (adapted from ISPW-6/7 benchmark example)

An **agent** is an actor with concrete, physical manifestations, such as a human individual. An agent has dependencies that apply regardless of what roles he/she/it happens to be playing. For example, if John, the project manager desires recognition from his boss, John wants the credit to go towards his personal self, not to the position of project manager (which he hopes to be filled by someone else upon his own promotion), nor to any of the abstract roles that John plays (e.g. "Monitoring Progress"). We use the term *agent* instead of person for generality, so that it can be used to refer to human as well as artificial (hardware/software) agents.

A **position** is intermediate in abstraction between a role and an agent. It is a set of roles typically assigned jointly to one agent. For example, the position of project manager **covers** the two roles of "Scheduling And Assigning Tasks", and "Monitoring Progress". We say that an agent **occupies** a position.

Figure 2 shows an example of an Actor Dependency model of a software engineering process organization with agents, roles, and positions. It is an adaptation of the ISPW-6/7 benchmark example [16]. The organization includes a project manager, design engineers and quality assurance engineers. The example setting includes six technical activities (from "Modifying Design" to "Unit Testing") and two management activities ("Scheduling And Assigning Tasks" and "Monitoring Progress") pertaining to the development and testing efforts required to respond to a change request.

The intentional structure of this organization closely resembles the one in Figure 1. Separating out the concepts of roles, positions, and agents gives a finer grouping of dependencies, so that one could identify more precisely how one dependency might lead to other dependencies. The dependency structure in Figure 2 can be understood in terms of three main systems. One set of dependencies can be traced to the customer's goal dependency to have a module changed. This leads to the project manager's dependency to have each portion of the project completed by the respective technical roles (shown at the bottom of the figure), and also to the dependencies among the technical roles. An "ISA" construct representing conceptual generalization/ specialization is used to simplify the presentation (near centre of figure).

A second system can be traced to the general manager's dependency on the project manager for no project overrun. This leads the project manager to depend on team members to be on schedule, and to notify completion. A third system can be traced to the general manager's dependency on the QA manager for high quality on software produced. This leads the latter's dependencies on the Reviewing and Testing roles. The remaining dependencies can be traced to the need for viability of the dependencies in the main systems. These will be discussed in section 4.

There can be dependencies from an agent to the position that it occupies. "Design Specialist" is a class of agents, each of whom having a dependency on the position that it occupies – namely "Design Engineer", for achieving the goal that designs produced be state-of-the-art. If the goal is not met, an agent may seek another position.

Roles, positions, and agents can each have subparts. Aggregate actors are not compositional with respect to intentionality. Each actor, regardless of whether it has parts, or is part of a larger whole, is taken to be intentional. Each actor has inherent freedom and is therefore ultimately unpredictable. There can be intentional dependencies between the whole and its parts, e.g., a dependency by the whole on its parts to maintain unity.

We use the term **association** to refer to a collection of roles, positions, and agents which are interconnected by the "plays", "occupies", and "covers" relations. Various specialized forms of associations can be defined by refering to their intentional properties. For example, part of the definition of a *team* might include the property that agents in the team are rewarded for their effort mainly at the aggregate level rather than at the individual level.

3 A conceptual modelling framework

The concepts of the Actor Dependency model are given formal interpretation to avoid ambiguity, and to permit the development of tools for manipulating knowledge expressed in the model and for drawing conclusions from them.

Actor Dependency concepts are defined in terms of more basic intentional concepts such as belief, goal, ability, and commitment. These concepts have been formalized in modal logic for modelling agents in AI (e.g., [2, 23]). We have adapted these for formalizing the dependency relationships between actors. For example, we characterize a committed dependency as a commitment on the depender side plus the depender's belief that the dependee is committed.

$$CommittedTo(a, b, \phi) \equiv DependerCommitted(a, \phi) \\ \wedge Believes(a, DependeeCommitted(b, \phi))$$

On the depender side, commitment implies that the depender believes that some plan will fail if the dependum (ϕ) is not achieved. This reflects the vulnerability aspect of the dependency.

$$DependerCommitted(a, \phi) \supset \\ Believes(a, \exists p(\neg\phi \supset fail(a, p)))$$

The enabling aspect of the dependency is reflected by the depender's belief that the dependee has a plan which will result in ϕ being true, and that everything that the plan depends on are viable.

$$DependeeCommitted(b,\phi) \supset$$
$$Believes(b, \exists p(result(p,\phi) \wedge allDepViable(b,p)))$$

These axioms are presented in detail in [27]. A preliminary version appeared in [24].

Software processes typically involve many roles, agents, and positions, with complex networks of dependencies. Embedding the Actor Dependency model into a formal conceptual modelling framework would allow the potentially large amounts of knowledge about software processes to be managed and used effectively. We have chosen to embed the concepts of the Actor Dependency model into the conceptual modelling language Telos [18]. In doing so, we obtain an object-oriented representational framework, with classification, generalization, aggregation, attribution, and time. The extensibility of Telos, due to its metaclass hierarchy and treatment of attributes as full-fledged objects, facilitates the embedding of new modelling features such as Actor Dependency concepts.

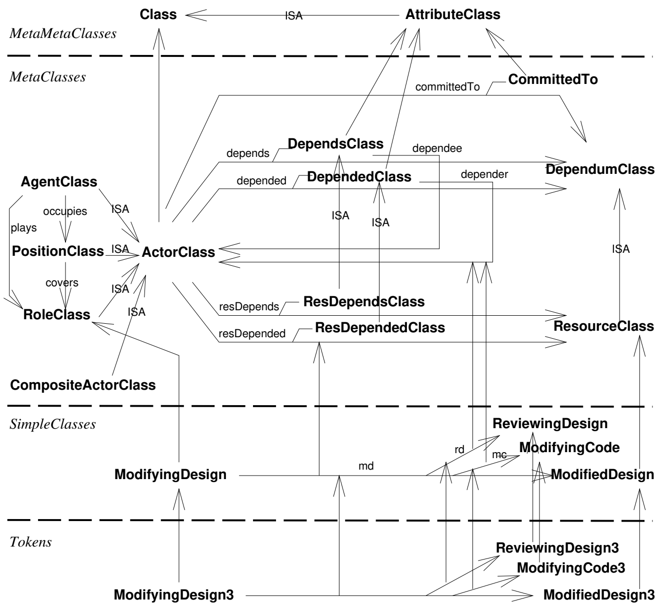

Figure 3: A partial semantic schema for the Actor Dependency model, with domain example (unlabelled arrows are InstanceOf links)

The schema for the Actor Dependency model is defined at the metaclass level in Telos (Figure 3). Domain classes such as `ModifyingDesign` would be defined as instances of some metaclass, in this case `RoleClass`. Metaclasses are instances of metametaclasses. Thus `RoleClass` and `ActorClass` are both instances of the metametaclass `Class`. This metaclass facility in Telos allows the schema to be expressed within the same framework as domain objects.

An instance of `ActorClass` (e.g., `ModifyingDesign`) can have as an attribute some instance of `DependedClass` (e.g., `ModifiedDesign`). This is used as the basic construct for representing actor dependencies. The schema for this is defined by the links labelled `depends` (if the named actor is the depender) and `depended` (if the named actor is the dependee).

We make use of the Telos facility for allowing attributes on attributes to specify the other party in a dependency. Since the attribute class `DependedClass` is a full-fledged object, we can define an attribute `depender` on it. The example shows `ModifyingDesign` as having two dependers (`ReviewingDesign` and `ModifyingCode`) on its dependum `ModifiedDesign`.

The four types of dependencies are defined as specializations on each of `DependsClass` and `DependedClass`. For brevity, Figure 3 only shows the specializations for Resource Dependency. Commitment is represented as another attribute on `ActorClass` with attribute value belonging to `DependumClass`. This can be used to qualify any dependency. Criticality is defined analogously.

Because Telos allows integrity constraints on any class, the semantics of the Actor Dependency model can be incorporated and enforced by stating them as integrity constraints in the appropriate metaclasses. Examples of the syntactic representation of the AD model in Telos are given in [26].

4 Analyzing software processes

A software process model that captures actors' motivations, intents, and rationales provides a better basis for an analyst to explore the broader implications of a process. Because software engineering activities involve uncertainty, actors need to be flexible enough to respond to contingent situations, and be prepared for setbacks. In acknowledging actors' freedoms and constraints, the Actor Dependency model permits richer types of analysis than conventional, non-intentional models. The formality of the AD model allows computational tools to be developed to support analysis. In this section we suggest some types of analyses by considering two important aspects of intentional dependency – the enabling aspect and the vulnerability aspect.

By enlisting the help of dependees, a depender expands opportunities, and can achieve what would otherwise be unachievable. The customer in the example of Figure 1 is able to have a system developed, by depending on the project manager, even if the customer has no ability to develop the system himself. The project manager does not have ability to development the system all by himself. He is enabled through dependencies on his technical team. Given an AD model of a software process, one could ask: What new relationships among actors are possible? By matching the open dependencies from dependers and dependees, one can explore opportunities that are open. Classification and generalization hierarchies facilitate the matching of dependums.

The down side of a dependency for a depender is that the depender becomes vulnerable to the failure of the dependency. A depender would be concerned about the viability of a dependency. Various mechanisms can contribute to fortifying a dependency and to mitigate vulnerability. In analyzing an AD model for viability of dependencies, we look for mechanisms such as *enforcement, assurance,* and *insurance*.

A commitment is *enforceable* if there is some way for the depender to cause some goal of the dependee to fail, e.g., if there is a reciprocal dependency. In Figure 1, each of the technical team members have dependencies on the project manager. These dependencies make the manager's dependency on team members enforceable. The customer's dependencies on the project manager are not directly enforceable, since there are no reciprocal dependencies. However, the general manager depends on the customer for payment and for good customer relations; and the project manager depends on the general manager for recognition. The customer's dependencies are therefore indirectly enforceable through the general manager. Each leg of indirectness introduces uncertainty and may weaken enforceability. The lack of dependencies from the project manager to the end-user (as opposed to the paying customer), either direct or indirect, would suggest that the user's dependencies on the manager are unenforceable. Figure 2 contains more examples of enforcement mechanisms. We note that enforcement loops often go through agents, since it is ultimately agents (especially human agents) who are vulnerable, not abstract roles or positions.

Another way to analyze viability of a dependency is to look for mechanisms for *assuring* commitment. Assurance means that there is some evidence that the dependee will deliver, apart from the dependee's claim. For example, knowing that fulfilling the commitment is in the dependee's own interest would be an assurance. In the example of figure 2, the professional standards and pride of QA specialists provide some assurance to the QA manager that his desire for maintainable software would be met. Unlike in enforcement-based measures, an assurance mechanism does not allow the depender to take action that can cause the dependee to correct its behaviour.

If a conflict of interest is detected, it would contribute negatively to the assurance of a dependency. In figure 2, the project manager depends on the design engineer not to add fancies features beyond the customer's requirements ("No Gold-Plating"). However, design specialists occupying the position of design engineer prefer to do state-of-the-art designs. This is negative assurance that the manager's no gold-plating dependency would be met. An analyst can use the AD model to analyze alignment of interests or conflicts of interests among various combinations of roles, agents, and positions.

Insurance mechanisms reduce the vulnerability of a depender by reducing the degree of dependence on a particular dependee. A depender can improve the chances of a dependum being achieved by having more than one dependee for the same dependum (or parts thereof). Including two software engineers from some other team to do design reviewing provides some *insurance* against failure by the development team to detect their own defects (in addition to addressing the problem of bias). Another type of insurance is the provision of extra resources to enable remedial or recovery action upon failure of the original dependency. Purchasing an insurance policy from an insurer is an example of this type. In contrast to enforcement or assurance, insurance measures can be taken on the depender side without involving the original dependee.

Measures for dealing with vulnerability are often taken in combination. A weekly status report might be used by the general manager to assure no project overrun, and as a basis for deciding whether and when enforcement action is necessary.

By analyzing the opportunities and vulnerabilities of actors, and the provisions that actors make to deal with vulnerabilities, an analyst can gain a fuller understanding of the "whys" behind a software process. Questions such as "Why do we need design reviews?", "Why does the review team have this membership composition?" and "Why does the general manager want weekly status reports?" can be answered more fully. An AD model provides the conceptual framework and the basis for analytical tools.

5 Designing software processes

The greater expressiveness of the AD model encourages process engineers to discern finer distinctions among process alternatives, and to choose among them based on their intentional characteristics. We illustrate with a small example. Figure 4 shows four alternative arrangements for accomplishing testing in a hypothetical organization.

Process design alternatives may be thought of as being organized into a tree (or, more generally, a graph). The figure shows two major branches, with the left branch consisting of three alternatives. All four alternatives meet the functional goal of `Completed(Testing)` (not shown), but are differentiated with regard to how well they meet non-functional process design goals (shown in the centre of the figure).

The relationship between the designing role and the testing role (and hence between designer and tester) are dif-

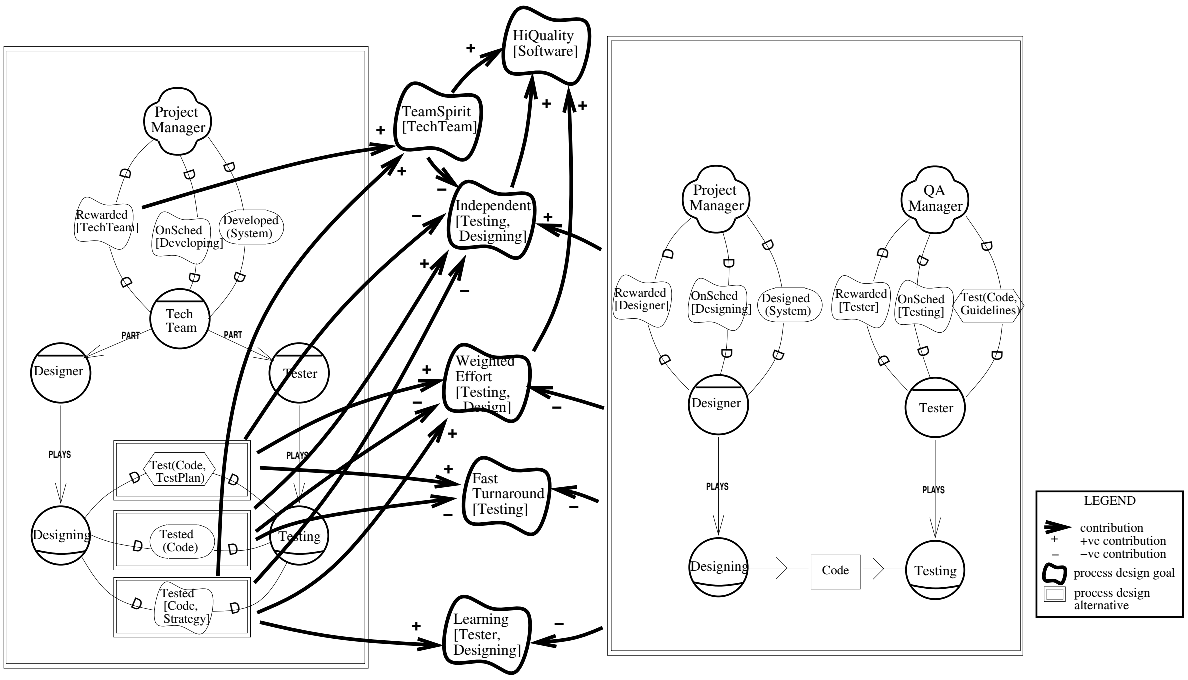

Figure 4: Four process design alternatives and their qualitative evaluation with respect to process design goals

ferent in the four alternatives. In the Task Dependency option (left-hand side, top), the designer tells the tester to follow a detailed test plan. This has the advantage of fast turnaround, but the disadvantage that no one other than the designer is really subjecting the software to test (negative contribution to the process design goal that the testing and designing roles should be independent).

In the Goal Dependency alternative (left, middle), the tester is given freedom on how to test, thus achieving some degree of independence. But testing would likely take longer to complete, and it would not be making use of knowledge about the design to focus testing on potential weak spots (negative contribution to `WeightedEffort[Testing, Design]`). A Soft-Goal Dependency has the advantage of making testing a cooperative venture, fostering team spirit, and contributing to the tester's learning about design.

Looking at the relationship between the project manager and technical team members, we see that the team is rewarded as a unit (on the left-hand branch), fostering a strong team spirit. This, however, makes all three alternatives on this branch weak with respect to independence of testing.

On the right-hand side, the designer and tester are rewarded separately for their efforts. Each have dependencies from their respective managers, who have no immediate dependencies between them. This alternative is good for achieving independence between testing and designing, but is negative for fast turnaround, design-weighted testing, and for the tester's learning about design.

The non-intentional nature of the flow of `Code` from designer to tester – as opposed to a Resource Dependency – indicates that the tester does not have goals that would be affected if the code is not received. This might be the case if the QA manager does not consider the responsibility of testing to begin until code is received. This is in contrast to the team situation on the left, in which the tester would be motivated to assure prompt completion of design, so that testing could begin on time, in order to assure reward for the whole team, of which the tester is part.

In studying a process and seeking ways to improve it, one could typically come up with many alternatives, possibly involving changes to human procedures as well as selecting among a variety of features in support environments. These alternatives need to be evaluated against many criteria including project objectives (such as quality and productivity) as well as personal concerns (such as reward structures and career paths). The process of designing software processes could be greatly facilitated by providing tools that can help process engineers to systematically pursue design goals by generating and analyzing alternatives, and to make tradeoffs. A framework for software process modelling, analysis, and design is suggested in the longer version of this paper [26], based on a proposed framework for requirements engineering [25].

6 Related work

In the Software Process Modelling research area, non-intentional models that focus on activities and input-output flow are the most common. More flexible formalisms include models with rules and triggers, and extensions of Petri nets (e.g., [5, 1]). These may be viewed as providing the "how", to better support or automate process enactment. The intentional model proposed in this paper focuses on the "why", in order to support reasoning about process improvement and redesign.

From the perspective of our framework, portions of software process models that are enacted by machine belong *inside* one (or more) of the "agents". The focus of the AD model is on external relationships between agents (human or otherwise). For computer-based agents, the AD model serves as a *requirements* level model. Further constraints are needed to reduce the requirements to a design specification, and from there to an implementation – expressed in a non-intentional representation such as procedural or Petri net formalisms [15]. Computer-based agents with planning and problem-solving ability (e.g., [10]), will require less reduction to reach an implementation.

This paper extends our earlier work [24, 25] by embedding the AD model in the Telos language, outlining some analytical concepts for use with the AD model, and further develops the concepts of role, position, agent, and association. Furthermore, by applying the framework to a complex environment, the power of the features of the AD model to capture subtle organizational issues are more fully illustrated.

Most requirements models of organizational environments employ some notions of entities, activities, and assertions, or variations, e.g., [9]. Concepts of goals, rules, methods, and tactics are used variously in a number of requirements frameworks, e.g., [7, 4, 8, 6, 19]. The distinguishing feature of the present framework is its introduction of Actor Dependency concepts. The motivation for these concepts comes from the area of organizational computing, where it has been recognized that computing system design needs to take into account the problematic and contingent nature of work [22]. The AD model accommodates uncertainty in organizational environments, and acknowledges actors' flexibility in coping with uncertainty, by not requiring design goals to be fully reduced to non-intentional activities and flows. The way organizations retain some degree of stability and structure is captured by means of intentional dependencies, reflecting actors' expectations on each others otherwise unpredictable behaviour. Expectations are not always met, so that *analyses* of enforcement, assurance, and insurance are of interest. A comparable concept of ensuring has been proposed in [4] for more controllable environments.

The AD model embodies a distributed conception of intentionality. The intentional dimension is represented as relationships between actors, with dependency chains propagating in all directions, criss-crossing the organization in the form of a network. This could be contrasted with an alternative conception of intentionality in which global organizational goals are hierarchically decomposed and assigned to individual agents. Similarly, design goals do not apply uniformly to all actors, but reflect the perspectives and interests of each stakeholder.

The concepts of role, position, agent, and association reflect how organizations group and manage complex patterns of social relationships. The need for multiple views is well recognized in requirements engineering (e.g., [20]). A view-directed strategy for requirements acquisition was suggested in [4]. The role/position/agent distinction suggests the possibility of role-centred, position-centred, and agent-centred design strategies, making use of the abstraction underlying the distinction which is reminiscent of layered independence in system architectures. The three strategies can potentially borrow design techniques from the areas of business design, job design, and human resource management, respectively.

7 Conclusions

In this paper, we have proposed a software process model that focuses on the intentional relationships among actors, and have outlined its use in the context of process analysis and design. The model is formal so that tools can be developed. The modelling concepts were illustrated with examples drawn from the software process literature [16]. We sketched how the formal properties of the model can be specified, and showed how it can be embedded in a conceptual modelling language. Analysis and design tools were illustrated by example, but remain to be implemented and tested.

Curtis et al. [3] have identified formality, granularity, and scriptiveness as important issues for software process modelling research. The Actor Dependency model is formal without being deterministic. Intentional concepts are used to model actors' expectations about each other's behaviour, and their provisions for unmet expectations. Acknowledging the inherent freedom of actors, analyses on the model focus on issues such as opportunities and risks. An intentional model places limits around an actor's behaviour, in terms of what is expected to be achieved, without explicitly specifying detailed process steps. This avoids the granularity dilemma encountered in non-intentional models, where a coarse-grained description is likely to underspecify by allowing too much freedom, while a fine-grained description tends to overspecify by including process characteristics that are circumstantial rather than essential. The AD model is primarily intended to be used in a descriptive mode. A prescriptive or proscriptive model which only specifies officially sanctioned or prohibited actions would not allow us to reason about the potential impacts of actors' *violations* of expectations and commitments, which we have illustrated in section 4.

This work represents an initial step in using intentional concepts to help understand and analyze software processes, and in the use of a requirements engineering approach to software process design. We have found that requirements engineering concepts can contribute towards a number of software process issues, and deserve further exploration. We believe that the modelling framework is particularly suited to software processes (as opposed to more general business processes) because of the often collaborative problem-solving nature of software work, the high complexity of the products, and the amenability of the work to computer tool support. However, the adequacy of the framework have yet to be tested in practice.

For future work, we need to integrate features of the AD model into Telos implementations, and to develop practical algorithms with tractable computational properties, so as to contribute towards a set of tools to aid in the systematic modelling, analysis, and design of software processes.

Acknowledgements. We thank the anonymous reviewers, L. Chung and B. Nixon for comments and suggestions that led to the improved presentation in this paper.

References

[1] S. Bandinelli and A. Fuggetta, Computational Reflection in Software Process Modeling: the SLANG Approach, *Proc. 15th Int. Conf. Soft. Eng.*, 1993, pp. 144-154.

[2] P. R. Cohen and H. J. Levesque, Intention is Choice with Commitment, *Artif. Intell.,* 42 (3), 1990.

[3] W. Curtis, M. I. Kellner and J. Over, Process Modelling, *Comm. ACM*, 35 (9), 1992, pp. 75-90.

[4] A. Dardenne, A. van Lamsweerde and S. Fickas, Goal-Directed Requirements Acquisition, *Science of Computer Programming*, 20, pp. 3-50, 1993.

[5] W. Deiters and V. Gruhn, Managing Software Processes in the Environment MELMAC, *Proc. 4th Int. Symp. Practical Software Development Environments*, Irvine 1990, SIGSOFT Notes 15, no. 6., pp. 193-205.

[6] E. Dubois, Ph. Du Bois and A. Rifaut, Elaborating, Structuring and Expressing Formal Requirements of Composite Systems, *Proc. Fourth Conf. Advanced Info. Sys. Eng.*, Manchester, U.K., May 12-15, 1992.

[7] M. S. Feather, Language Support for the Specification and Development of Composite Systems, *ACM Trans. Prog. Lang. and Sys.* 9, 2, April 1987, pp. 198-234.

[8] S. Fickas and R. Helm, Knowledge Representation and Reasoning in the Design of Composite Systems, *IEEE Trans. Soft. Eng.*, 18, 6, June 1992, pp. 470-482.

[9] S. J. Greenspan, *Requirements Modelling: A Knowledge Representation Approach to Software Requirements Definition,* Ph.D. Thesis, Dept. Comp. Sci., Univ. of Toronto, 1984.

[10] K. E. Huff and V. R. Lesser, A plan-based intelligent assistant that supports the software development process, *Proc. 3rd Symp. Practical Softw. Dev. Envs., Soft. Eng. Not.* 13(5) 1989, pp.97-106.

[11] W. Humphrey, *Managing the Software Process*, Addison-Wesley, Reading, Mass., 1989.

[12] *Proc. 8th International Software Process Workshop*, 1993.

[13] *Proc. 2nd International Conference on the Software Process*, Berlin, Germany, Feb. 1993.

[14] *Proc. 15th Int. Conf. Soft. Eng.*, Baltimore, May 1993.

[15] M. Jarke, J. Mylopoulos, J. W. Schmidt, Y. Vassiliou, DAIDA: An Environment for Evolving Information Systems, *ACM Trans. Info. Sys.*, 10(1) Jan. 1992, pp.1-50.

[16] M. Kellner, P. Feiler, A. Finkelstein, T. Katayama, L. Osterweil, M. Penedo, and H. Rombach, Software Process Modeling Example Problem, from *7th Int. Software Process Workshop*, Yountville, California, Oct. 1991.

[17] N. Madhavji, The Process Cycle, *IEE Software Engineering Journal*, Spec. Issue on Software Process and Its Support, N. Madhavji, W. Schäfer, eds., 6(5) Sept. 1991, pp.234-242.

[18] J. Mylopoulos, A. Borgida, M. Jarke, M. Koubarakis, Telos: Representing Knowledge about Information Systems, *ACM Trans. Info. Sys.*, 8 (4), 1991.

[19] J. Mylopoulos, L. Chung, B. Nixon, Representing and Using Non-Functional Requirements: A Process-Oriented Approach, *IEEE Trans. Soft. Eng.*, 18 (6), June 1992.

[20] B. Nuseibeh, J. Kramer, A. Finkelstein, Expressing the Relationships Between Multiple Views in Requirements Specification, *15th Int. Conf. Soft. Eng.*, Baltimore, 1993, pp.187-196.

[21] C. Potts and G. Bruns, Recording the Reasons for Design Decisions, *Proc. Int. Conf. Soft. Eng.*, 1988, pp. 418-427.

[22] L. Suchman, Office Procedures as Practical Action: Models of Work and System Design, *ACM Trans. Office Info. Systems*, 1(4) Oct. 1983, pp.320-328.

[23] B. Thomas, Y. Shoham, A. Schwartz, and S. Kraus, Preliminary Thoughts on an Agent Description Language, *Int. J. Intell. Sys.*, Vol. 6, 1991, pp. 498-508.

[24] E. Yu, Modelling Organizations for Information Systems Requirements Engineering, *Proc. 1st IEEE Symp. on Requirements Engineering*, San Diego, Jan. 1993, pp.34-41.

[25] E. Yu, An Organization Modelling Framework for Information Systems Requirements Engineering, *Proc. 3rd Ws. Info. Techs. & Sys.*, Orlando, Dec. 1993, pp. 172-179.

[26] E. Yu and J. Mylopoulos, *Understanding "Why" in Software Process Modelling, Analysis, and Design*, Tech. Report DKBS-TR-94-3, Dept. Comp. Sci., Univ. of Toronto, Feb. 1994.

[27] E. Yu, *A Framework for Organization Modelling*, Ph.D. Thesis, Dept. Comp. Sci., Univ. of Toronto, forthcoming.