Handbook of Research on Socio-Technical Design and Social Networking Systems

Brian Whitworth
Massey University-Auckland, New Zealand

Aldo de Moor
CommunitySense, The Netherlands

Volume I

Information Science REFERENCE

INFORMATION SCIENCE REFERENCE
Hershey • New York

Director of Editorial Content:	Kristin Klinger
Director of Production:	Jennifer Neidig
Managing Editor:	Jamie Snavely
Assistant Managing Editor:	Carole Coulson
Typesetter:	Michael Brehm
Cover Design:	Lisa Tosheff
Printed at:	Yurchak Printing Inc.

Published in the United States of America by
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue, Suite 200
Hershey PA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

and in the United Kingdom by
Information Science Reference (an imprint of IGI Global)
3 Henrietta Street
Covent Garden
London WC2E 8LU
Tel: 44 20 7240 0856
Fax: 44 20 7379 0609
Web site: http://www.eurospanbookstore.com

Library of Congress Cataloging-in-Publication Data

Handbook of research on socio-technical design and social networking systems / Brian Whitworth and Aldo de Moor, editors.
p. cm.
Includes bibliographical references and index.
Summary: "Every day throughout the world, people use computers to socialize in ways previously thought impossible such as e-mail, chat, and social networks due to emergences in technology. This book provides a state-of-the-art summary of knowledge in this evolving, multi-disciplinary field"--Provided by publisher.
ISBN 978-1-60566-264-0 (hardcover) -- ISBN 978-1-60566-265-7 (ebook)
1. Online social networks. 2. Internet--Social aspects. 3. Information technology--Social aspects. I. Whitworth, Brian, 1949- II. Moor, Aldo de.
HM742.H37 2009
303.48'33--dc22

2008037981

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

.

Chapter XX
Formal Analysis of Workflows in Software Development

Harry S. Delugach
University of Alabama in Huntsville, USA

ABSTRACT

Automated tools are often used to support software development workflows. Many of these tools are aimed toward a development workflow that relies implicitly on particular supported roles and activities. Developers may already understand how a tool operates; however, developers do not always understand or adhere to a development process supported (or implied) by the tools, nor adhere to prescribed processes when they are explicit. This chapter is aimed at helping both developers and their managers understand and manage workflows by describing a preliminary formal model of roles and activities in software development. Using this purely descriptive model as a starting point, the authors evaluate some existing tools with respect to their description of roles in their processes, and finally show one application where process modeling was helpful to managers. We also introduce an extended model of problem status as an example of how formal models can enrich understanding of the software development process, based on the analysis of process roles.

People sometimes make errors. The problem here was not the error, it was the failure of NASA's systems engineering, and the checks and balances in our processes to detect the error. That's why we lost the spacecraft.

Edward Weiler,
NASA's Associate Administrator on the loss
of the $327 million Mars Climate Orbiter.

INTRODUCTION

Many automated tools are available to support software development. There are two main reasons for an organization to use these tools:

- Much of software development takes place in distributed environments, or at least where the participants have difficulty meeting regularly face-to-face. Automated (often web-based) tools allow them to collaborate in a generally cost-effective way compared to travel and shipping costs.
- Software development workflows prescribe various *activities* to be tracked and *artifacts* to be created and maintained. Even when developers are able to collaborate in person, the number of these can become large and therefore requires organizing tools and a central repository.

As with all tools, their effectiveness is determined by how well participants understand how to use them. There is ample evidence that mere use of tools is not sufficient to support an effective workflow. Even if developers understand a tool's basic operation, they often do not understand or adhere to any development process supported (or implied) by the tool. This chapter examines some popular web-based software engineering tools from a pragmatic role-oriented perspective. That is, we intend to focus on the roles and purposes within the context of the development process, rather than characteristics of artifacts or products.

Our ultimate goals in developing these models is the following:

1. To better describe and analyze the processes themselves.
2. To formally analyze and evaluate tools with respect to generally accepted process models, and
3. To formally compare and contrast the models with each other.
4. To provide formal definitions based on process models.

The approach in this chapter illustrates all four of these goals. First we motivate the general value of formal models in analyzing process, and then provide some background on workflow modeling with respect to the software development process. The main body of the chapter applies this approach to one particular sub-process (namely bug tracking). Each of the four goals is discussed in turn, using examples to illustrate the approach.

This work continues in the spirit of previous work in modeling development processes (Delugach, 2007) (de Moor & Delugach, 2006) and in using conceptual graphs for modeling communication (Delugach, 2006) and software development (Delugach, 1996) (Delugach, 1992). In this chapter, we use conceptual graphs—a well-known knowledge representation—as a clear and effective way of formally representing the parts of a workflow. In future work, some automated analysis may use conceptual graphs' formal basis in logical reasoning and inference; however, this chapter does not exploit those capabilities for these illustrations.

THE VALUE OF FORMAL MODELING

At this point, it is useful to evaluate the role of formal modeling in software system development. While nearly all developers acknowledge the value of formally modeling the software system itself, there is less agreement on the role of formal modeling of the *process* of software development. Resistance to this idea is usually caused by "horror stories" of:

- Incorrect or incomplete models of a process, which initially give the impression of soundness but then later reveal themselves to be inappropriate
- Models that have been imposed on a development team by either upper management without proper evaluation, or by contractual obligations that are included as "boiler plate" requirements without any evaluation as their appropriateness

- Models that are perceived as reducing someone's power or control, infringing on their "turf" or responsibilities to the overall project.

In short, using the wrong model is probably worse than having none at all. Many (most?) of these bad experiences stem from misunderstanding the proper role of a model. This chapter promotes descriptive models: these do *not* impose or require a particular structure or process, but serve as a template for evaluating processes and comparing them with each other.

One feature of the formal models proposed in this chapter is to describe roles and define them in terms of responsibilities with respect to the workflow. As the reader will see, one such role is the responsibility for fixing a software problem that has been previously identified. Many tools exist for dealing with defects in software, but organizations often do not have a repeatable process for applying them. That doesn't mean organizations can't succeed without them, but it does suggest that the personnel may not always know how they were able to succeed. Use of a formal model can help them understand.

Here is a typical response when such a model is proposed for an organization's evaluation:

It's a waste of time to define roles and responsibilities. There's no formal way to decide who fixes a bug—the matching of an individual to a problem is dependent upon the nature of the defect. A well-established team avoids that issue as that allocation process happens informally. What's the problem?

This attitude deserves a detailed response.

First of all, the "waste of time" idea deserves further study. Formal modeling takes time and resources, like any other software development support activity. Certainly tracking those resources will help over time to determine whether such approaches are cost-effective—such studies must be performed and their conclusions verified. Such studies are outside the scope of this chapter, however.

Next, the models shown in this chapter do *not* describe a formal way to decide who fixes a defect. Since a defect is usually characterized by some general attributes, these might be formally matched up to known developer skills. (Some of these skills might not in fact be generally known; "did you know that person Y used to work on demographic databases?") We would never propose that developers blindly or arbitrarily follow a model, any more than a taxi driver with a GPS navigator should drive through a "road closed" barrier. We do, however, propose that the model can provide guidance to managers and developers if they choose to be guided.

The last claim about "a well established team" is an interesting one. Over the long term, intact teams of experienced people tend toward informality – either their once-formal procedures have become internalized or else they depend upon trust and past experience to guide their (informal) interactions. This is effective in some groups, and thus provides seeming counter-examples to the claim of formality's usefulness; however, trust and past experience usually require long periods of interaction that not all teams are fortunate enough to possess. In many distributed development environments, developers have not ever met face to face. Personnel may come and go, further interfering with the effect of experience and trust. In short, formal models of a process can help current and future participants to understand what their responsibilities are, as well as understanding others' responsibilities as well.

We are familiar with this last claim. In fact, in one of our studies, we formally modeled a team process in consultation with the manager of the team (de Moor & Delugach, 2006). This was a "well established" team, some of whom had been working together for a few years. The model revealed that in a small team, when personnel fulfill more than one role, it is possible for checks and balances to be circumvented if the same person fulfills both the executing role and the evaluating role. This situation represented a potential conflict of interest in the team (see below) that the manager didn't realize and responded with "I think I want to look into that one."

Failures in software system development (an important group of technical systems in general) are well documented. Here we are not talking about defects in the software itself, but shortcomings in the development processes themselves. Spectacular failures appear in public news reports from time to time. One well-documented failure was the NASA's Mars Climate Orbiter in 1999 in the United States (MCO-MIB-I, 1999). The spacecraft flew too close to Mars and was permanently lost, a cost of $327.5 million. The spacecraft's trajectory was wrong because software teams in two different locations made different implicit assumptions about which measurement units were used. Although each team's calculations were completely correct (presumably to a number of decimal places), one team assumed metric (SI) units, while the other team assumed English units.

The point of this failure is not that simple mistakes can have profound effects. The investigation of the failure showed that the error was evident every time the spacecraft did a course correction with its rocket engine, but the personnel monitoring the spacecraft simply expected another correction to fix it. There were thus two project failures—the first committed by the distributed development team in its inability to detect the inconsistency, and the second committed by the monitoring team in not understanding that they were observing unexpected behavior and not knowing where or how to report it.

Both of the failures could have been detected by modeling, if the distributed teams had been able to share each other's model of both the software's development itself and the management of the project during flight; however, no such modeling occurred. This chapter argues that the modeling approach supports analysis of such possible errors before they happen. Of course, there is no guarantee that any approach will solve this problem, but without doing something, the problem will persist. One prominent software engineer writes about "... the difficulty of technology transition and the cultural change that accompanies it. Even though most of us appreciate the need for an engineering discipline for software, we struggle against the inertia of past practice and face new application domains (and the developers who work in them) that appear ready to repeat the mistakes of the past." (Pressman, 2001, p. 870).

The next section describes the approach for workflow modeling.

WORKFLOW MODELING

This chapter is intended to provide an approach for describing and evaluating software development processes, while focusing on two particular examples. It is important to emphasize the purpose of modeling is not to impose structure on an existing process, but to help understand what the structure is. Our position is therefore neutral with respect to being either normative or descriptive (i.e., neither "to-be" or "as-is" in the sense discussed by Scacchi (Scacchi, 2002)). While the discussion in (Scacchi, 2002) gives valuable insight into an environment (namely, open-source) where prescriptive models may not be viable or useful, this chapter takes the position that one must first have a model of a workflow in order to effectively understand, evaluate and ultimately improve that workflow. There are probably many uses of the model once an organization has produced them.

It is useful at this point to briefly mention some other processes for general problem-solving, both for comparison purposes as well as a reminder that process modeling (especially for problem solving) is not new; considering it from the general point of view may provide some insight. Some techniques are described in (Levinson & Rerick, 2002). Among some well-known problem solving processes are the Team-Oriented Problem Solving (TOPS) process pioneered by Ford Motor Company in the USA. This approach is sometimes called the "8-D" approach because of the eight "dimensions" it is intended to address; these are described simply as steps in the process.

Figure 1 is a typical description of the process (verbs italicized for emphasis).

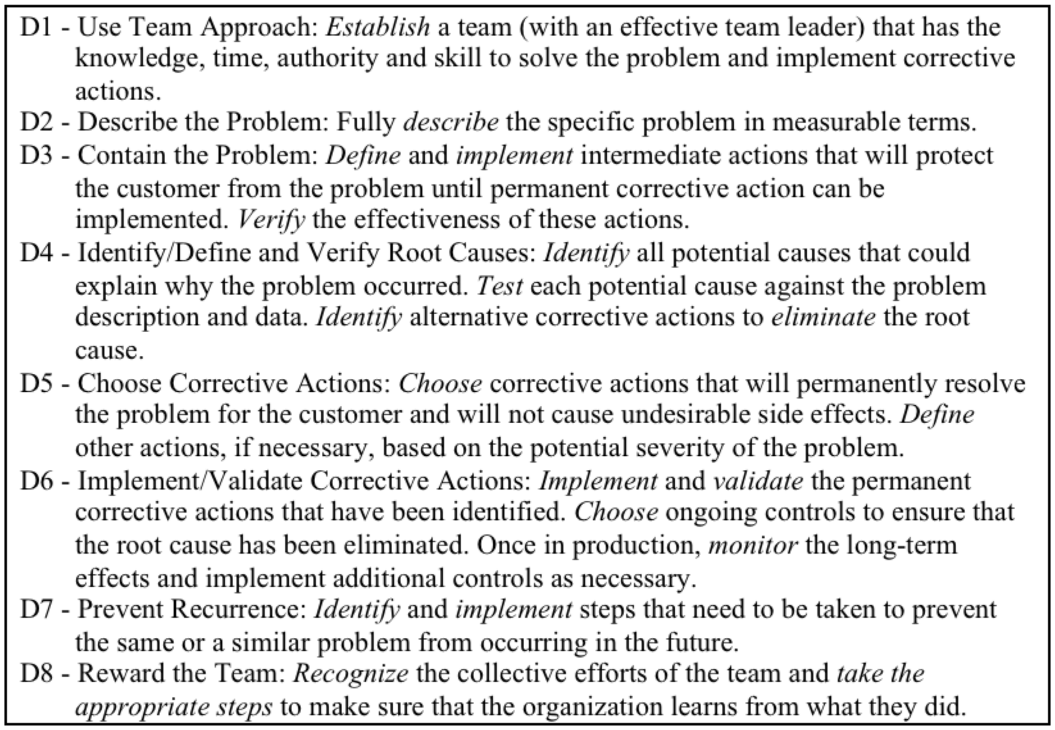
An important point, which will be made several times during this chapter, is that the process focus is usually on particular activities or tools, without a clear delineation or definition of roles. For example, it is often useful to have independent (or at least different) personnel do the monitoring and validating of another group's constructive activities. Though there are several activities identified (e.g., *define*, *implement*, etc.) that are required to be performed by a TOPS team as a whole, there is no particular guidance (at this level of the description) for *who* on the team will perform those activities, even given in D1 that someone is presumed to have the knowledge, time, authority or skill. There is also no clear idea of how the process should be monitored or audited for being carried out correctly or effectively.

Some may argue that each organization should prescribe its own process for assigning these roles, monitoring, auditing, etc. We agree that organizations should do that; our approach does not prescribe any particular process development methodology. Their own methodology must answer the question: what's the right process for a given organization? Our approach is meant only to (i) alert organizations to possible omissions in their process descriptions (whether they choose to fill them in or not), (ii) suggest some typical roles if they do choose to specify them and (iii) suggest the value of being able to analyze and improve processes that are supported by explicit models.

(Levinson & Rerick, 2002) mentions other problem solving techniques (e.g., the "Deming wheel" of Plan-Do-Check-Act (PDCA) and Six Sigma's DMAIC) whose descriptions similarly omit a clear description of roles at the top level. Of course, refinements of these techniques have provided more guidance about the roles needed, but again they are considered secondary to the primary process framework.

Being "informal" does not render a process incapable of being modeled; on the contrary, most formal processes have their origins as informal activities that underwent successive refinement. We begin with the assumption that developing any model of a process is generally more useful than not modeling it at all. That being said, this chapter does not propose imposing any particular model on any software development environment; rather

Figure 1. TOPS problem-solving process



D1 - Use Team Approach: *Establish* a team (with an effective team leader) that has the knowledge, time, authority and skill to solve the problem and implement corrective actions.

D2 - Describe the Problem: Fully *describe* the specific problem in measurable terms.

D3 - Contain the Problem: *Define* and *implement* intermediate actions that will protect the customer from the problem until permanent corrective action can be implemented. *Verify* the effectiveness of these actions.

D4 - Identify/Define and Verify Root Causes: *Identify* all potential causes that could explain why the problem occurred. *Test* each potential cause against the problem description and data. *Identify* alternative corrective actions to *eliminate* the root cause.

D5 - Choose Corrective Actions: *Choose* corrective actions that will permanently resolve the problem for the customer and will not cause undesirable side effects. *Define* other actions, if necessary, based on the potential severity of the problem.

D6 - Implement/Validate Corrective Actions: *Implement* and *validate* the permanent corrective actions that have been identified. *Choose* ongoing controls to ensure that the root cause has been eliminated. Once in production, *monitor* the long-term effects and implement additional controls as necessary.

D7 - Prevent Recurrence: *Identify* and *implement* steps that need to be taken to prevent the same or a similar problem from occurring in the future.

D8 - Reward the Team: *Recognize* the collective efforts of the team and *take the appropriate steps* to make sure that the organization learns from what they did.

it is an attempt to establish a framework for modeling and thence understanding one's own software development environment, especially with respect to the roles that necessarily appear in any human-supported workflow. As the range of environments becomes more diverse (e.g., open source, agile methods, etc.) it becomes even more important to develop models and then to validate them.

Workflow modeling, as used in this chapter, is taken from the workflow specification definitions used in (de Moor & Jeusfeld, 2001), and based on the RENISYS model of organizational roles (de Moor, 1997). One key feature of those models is the notion of organizational actors, each of whom has particular obligations with respect to their roles in various activities. We use conceptual graphs (Sowa, 1984) as a convenient formalism and easily understood visual aid to represent the models. Conceptual graphs are well described elsewhere (Sowa, 1984) (Polovina & Heaton, 1992).A simple model of a workflow activity is shown in Figure 2, adapted from (de Moor & Delugach, 2006).

The graph in Figure 2 may appear simple; in fact, we consider this one of the strengths of conceptual graphs. This chapter proposes some modifications and additions to this model, based on some shortcomings in its power to express some important pragmatic relationships in the bug tracking process.

We will focus on the workflow involved in two different software development activities: problem reporting (often called "bug tracking") and requirements change. Software problem tracking is one portion of a much large set of processes belonging to software configuration management (SCM) which has been extensively studied; for a summary, see (Pressman, 2001). The motivation for a controlled SCM process comes from the observation that software systems constantly change while under development, either through additional requirements or business needs, or through the natural process of successively refining artifacts from inception to deployment. Because this process has many purposes, there are often many people involved.

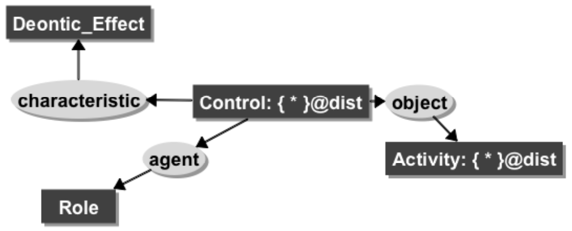
Another software activity is the process of making a change to the formal requirements in a software project. Along with de Moor, we conducted an experiment in modeling such a process in an industrial software development setting (de Moor & Delugach, 2006). This process is important because of the large lifecycle costs that result from erroneous requirements, a well-documented phenomenon (Pressman, 2001). Although requirements change is likewise subject to the control of SCM, we do not specifically address that feature in this chapter.

This chapter is focused on modeling software engineering processes with respect to its role- and purpose-oriented human aspects: who is involved, what are stakeholders' roles in the process's success, what responsibilities do they hold with respect to the system and what goals are they expected to pursue. The intent is that developers will better understand the processes they are using, perhaps finding omissions and mistakes along the way. We have applied these techniques in a production-level environment and as a result were able to suggest improvements to an organization's activities within a specified process.

MODELING SOFTWARE DEVELOPMENT WORKFLOWS

The main emphasis of this chapter is to formally capture software development workflows so that

Figure 2. Workflow step represented by conceptual graphs



There is a set of control concepts, each of which is characterized by a deontic effect (see below) and each performed by a particular role. For each control concept, there is a set of activities, each of which is operated upon by that control.

they may be better understood and analyzed. This section suggests the main area where current workflow descriptions are lacking. We will restrict our current attention to one task in software engineering; namely, bug tracking. The goal of this section (and of this chapter) is not restricted to this particular process, however; it is our way of illustrating the various uses of formal models in general.

This section illustrates the four goals given in the introduction. First we show how models can be used to describe software engineering processes. Next we show how models can be used to analyze particular software engineering processes and tools for their completeness and understandability. Then we show how models can be used to compare models to one another, in this case a prescribed process model vs. a model of the actual practice. Finally we offer an example of how definitions can be created from formal process models.

Describing Software Engineering Workflows

This section illustrates how formal models are used to describe software engineering workflows. We chose bug tracking as a typical activity in software engineering.

Bug tracking can be viewed as one kind of *problem resolution process.* The software engineering community has established standards for such processes, as exemplified in ISO/IEC 12207 (ISO/IEC, 1996). In this section of the chapter, we first describe some generic problem resolution process steps using the workflow models developed previously, then we briefly describe the 12207 process, and summarize the bug tracking processes supported by Bugzilla, a well-known software development tool set. Although primarily known as a tool (not a methodology), its description implies a process to be followed when using the tool's bug tracking features. In this section, formal models are shown for the ISO/IEC 12207 process and Bugzilla.

The first thing that one notices in studying the bug tracking capabilities of existing tools and processes is that there is generally no explicit set of roles which are defined in the process. Of course, the mere existence or use of a tool never guarantees that it will be used effectively or even correctly; however, most tools seem implicitly geared toward a particular change control process. Some of them appear to imply certain roles, while others appear role-neutral.

Requirements modification may also be seen as a problem resolution process, but of a different sort. In bug tracking, problems are explicitly identified by testers or users; in requirements modification, problems are usually identified through analysis. In either case, once the problem is identified, certain steps are performed in the resolution of the problem. In most descriptions of workflows, some key pragmatic knowledge is either left implicit or not even considered. This chapter's approach models the workflow so that some of that implicit pragmatic knowledge can be filled in.

First we will illustrate the kind of knowledge that is often omitted in descriptions of workflows, even when they appear to be well defined. The tools in sourceforge, for example, include a bug tracker. A tracked bug using sourceforge's tracker has the following attributes: assignee, status, category, group and description. Note that few of the attributes have any reference to persons or roles' responsibilities in software development.

- **Assignee:** The project administrator to which a tracker item is assigned. Can be chosen from one of the administrators registered in this project.
- **Status:** This is the (potentially changing) current status of a bug. The online help says:
  You can set the status to 'Pending' if you are waiting for a response from the tracker item author. When the author responds the status is automatically reset to that of 'Open'. Otherwise, if the author doesn't respond within an admin-defined amount of time (default is 14 days) then the item is given a status of 'Deleted'.
  This provides the beginnings of a primitive set of definitions for the possible status values,

and perhaps implies a particular workflow, however unstated.

- **Priority:** a nine-level scale.
- **Category:** "project-specific".
- **Group:** "project-specific".

The list of sourceforge's bug attributes clearly illustrates one of the major hurdles for practitioners in developing systems using existing tools: there is no structure or process guidance provided! To be sure, sourceforge's organizational goal is not to develop or impose specific processes, so one of its goals is to ensure as much flexibility as possible. Our approach likewise does not *require* completeness or assess quality; our main purpose here is to show how the approach can be used to analyze and evaluate different specified processes.

The attributes of "category" and "group" are good examples of this: each project administrator can choose them based on their own preferences. The downside of this approach is that the automated bug tracker has no capability to relate them to each other, to accommodate constraints between particular categories, groups or values of the other attributes (except for the ability to search each list by value). For example, are "category" and "group" orthogonal to each other, or is a group a sub-category, etc.?

We point out that software development is not alone in lacking clear organizational responsibilities for various activities in a process. This section describes our model of an organizational process (including an ontology) and later we will show how to model roles in a formal way. As an illustration, we give general models for two bug-tracking activities: reporting and repairing.

We adopt Figure 8 as a description of a general *workflow step* with some pragmatic knowledge. Note the inclusion of the concept **Intention** with respect to a role in the process. This concept is lacking in previous models, which simply showed the obligations (required, allowed, prohibited) as the deontic effect assigned to a particular role. Previous models therefore did not give any indication as to *why* a particular role would be given a particular assignment.

For example, why would a program manager be required to review a change, or why would a developer be allowed (but not required) to make a change? For our future goals, if we want to reason automatically about roles and their appropriateness or legitimacy, we must start to model their purpose and relationship to the system's development as a whole. Figure 8 shows a more complete model of a workflow step. While the language is not great prose, it captures the essential elements of what the graph says. More importantly, the graph itself is formal and we can therefore reason about it automatically.

The model in Figure 3 is meant to emphasize that a participant's intentions need to be captured for each activity in a workflow model, as well as the status intended for the result(s) of that activity. (Later, we will propose a more formal idea of what "status" really means.)

Figure 4 shows a basic ontology for the bug-tracking domain. Arrows represent the supertype-of or is-a-kind-of relationship, in the taxonomic sense. For example, an **Activity** is a kind of **Process**, **Approve** is a kind of **Activity**, and so on. The "QA" role represents that of Quality Assurance, whose duties include (among other things) verifying that processes have been followed. This ontology is taken from (Delugach, 2007). This constitutes a summary of the full ontology's description; all of the types require definitions, which can be represented using conceptual graphs. One important point of such an ontology is to make a clear distinction between roles, intentions and obligations (deontic effects). In some organizations, these are lumped together in such a way that they are difficult to understand and therefore difficult to adapt for new workflows and situations.

Figure 5 and Figure 6 illustrate how to model two typical bug-tracking steps. The point here is that a formal model can help developers visualize their process, remind them of their obligations and also allow process analysts to compare different models to each other, process vs. practice models, etc.

Note in Figure 5 that the bug report is both a result of the initiated request and a goal of the developer's report activity. This may seem obvious:

why would the developer start something if they didn't have its result as a goal? Our point in this chapter is that process descriptions may implicitly assume this, but they either omit the goal, or the role, or both. Models can help identify "obvious" omissions, leaving it to the organization as to what to do with them.

Figure 6 shows a general template for the process of fixing a bug.

Our motivation for establishing basic graph models for these workflows stems from a belief that by analyzing them, we can identify potential missing or incorrect elements in existing workflows. Once the graphs are established, it is necessary to validate them. One avenue of validation would be to use conceptual graph tools to scan the wealth of existing data as advocated in (Ripoche & Sansonnet, 2006). Examples of natural language sources are emails, forum posts, program source code comments (Etzkorn, Davis, & Bowen, 2001), and even identifier names in programs (Etzkorn & Delugach, 2000). The task of validating graphs linguistically is a significant one, but beyond the scope of this chapter. We confine ourselves to providing simple English paraphrases of the graphs as an aid to understanding and potential validation.

Analyzing Particular Software Engineering Workflows and Tools

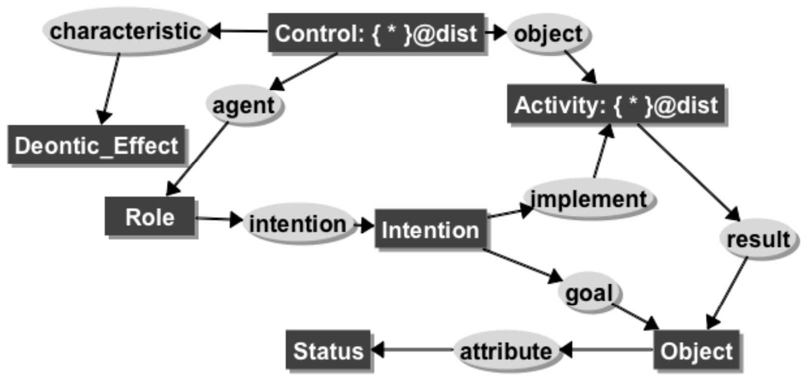
We have already suggested some important omissions in typical process models. This section illustrates those omissions by showing models of two bug-tracking processes using the conceptual graph representation already introduced. We then lead into a discussion of sourceforge's *status* indicator, as a typical example of an underspecified attribute for the purposes of process support.

ISO/IEC 12207 Problem Resolution Process

The problem resolution process of the ISO/IEC 12207 standard is reprinted in Figure 7.

This standard's process description is shown so that the reader can note one striking omission: nowhere does it prescribe *who* is tasked with any of the steps or activities! For example, the standard says "analysis shall be performed" but it does not state who will perform the analysis. This lack of specified roles weakens an organization's ability to provide appropriate process descriptions, including specifying who does what and also providing reasonable checks and balances for management. (Incidentally, some technical writers recommend using "passive voice" which ends up encouraging the lack of role knowledge.)

Figure 3. Workflow model incorporating intention



There is a role with some level of control ("deontic effect") over some activity that the role intends to implement. The activity's result is an object that is the goal of the role's intention. The object has a status.

Figure 4. Ontology for a role-based analysis of bug tracking

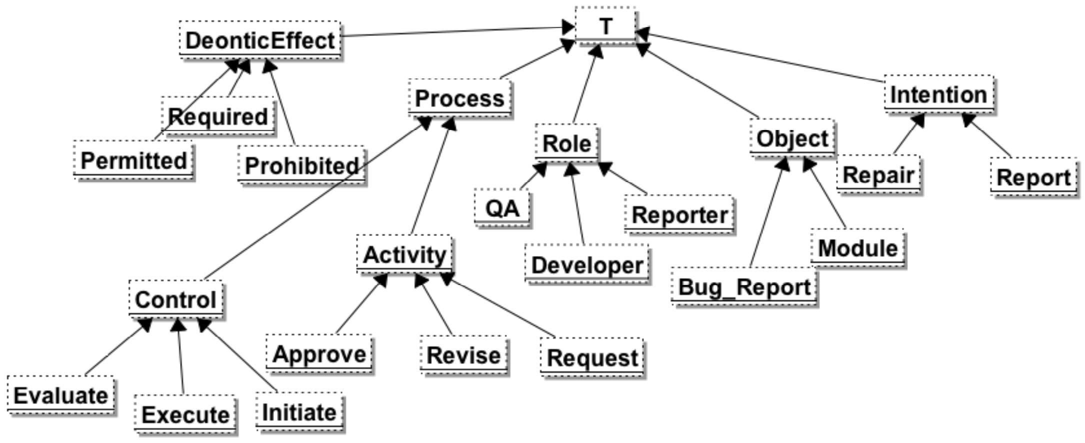
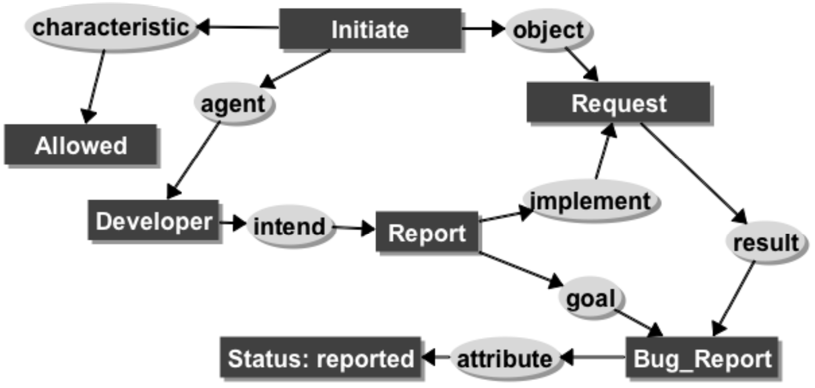
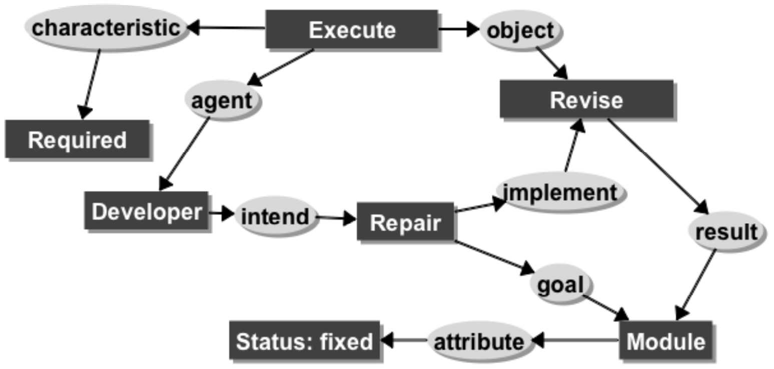


Figure 5. Generic model for reporting a bug



There is a developer who is allowed to initiate a request whose result is a bug report with status "reported". The same developer intends to report the bug report using the request.

Figure 6. Generic model for fixing a bug



There is a developer who is required to execute a revision whose result is a module with status "fixed". The same developer intends to repair the module using the revision.

The ISO/IEC 12207 process's model of problem resolution is shown in Figure 8. Compare this graph to the one in Figure 3. Note that while the deontic effect of "Required" is present (meaning initiation is required), there is no role shown that is responsible for that initiation, nor is there any indication of the purpose of the problem report or the goal in "handling" it. In short, the model is clearly incomplete, in ways that could directly impact an organization's ability to understand the process and therefore to implement it reliably in their workflow or audit its correct implementation.

Bugzilla

The Bugzilla bug tracking process is described in Figure 14 (taken from Figure 6-1 of the Bugzilla Guide at http://www.bugzilla.org/docs/3.0/html/). Note that several of the transitions have no labels, indicating that while it is possible for a bug to follow that transition, there are no constraints on when or how that transition is permitted. As in most other descriptions of these kinds of workflows, there is little guidance as to *who* is authorized to change the status of a bug. One might assume that the "owner" of a bug is authorized to change its state, but even in that case there is little organizational support for the reasons or circumstances under which the change is legitimate. For example, what does "unconfirmed" mean? The owner could simply mark a bug as "unconfirmed" if they did not want to deal with it at the moment, or the owner could engage in a detailed exploration and be unable to reproduce the bug, or perhaps the owner just hasn't had time to check out the bug yet.

In short, participants in a given software development workflow need a set of guidelines, constraints, operating procedures, etc. that govern what these status values mean. In a more sophisticated process, there would be procedures for changing/augmenting the set of status values as the team gains more experience.

The Bugzilla process is somewhat more completely defined than in the ISO/IEC 12207 process. Using Figure 9 as a basis, we can describe the model formally as shown in Figure 10. Again compare this graph to the one in Figure 3.

Note that the Bugzilla model, while still rather informal, does in fact include much of the vital pragmatic knowledge needed for an organization to implement the process. Roles are shown in several places, and verbs indicated activities are also shown. "Ownership" and "possession" are not specifically represented in the process models, but does seem to suggest a "required" obligation of some sort. In summary, Bugzilla's process appears more complete than the ISO/IEC one in.

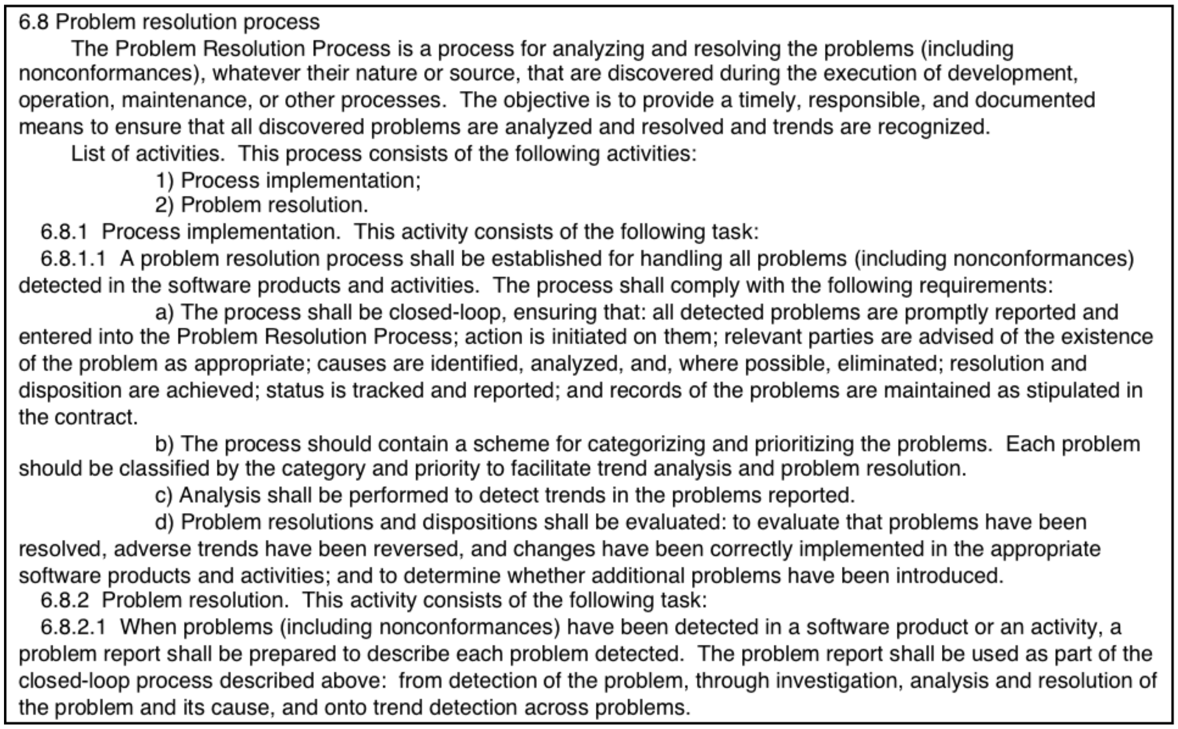
This section showed clearly the lack of role and goal knowledge in workflow descriptions, as well as illustrating the need for such knowledge. Again, it is not the purpose of this chapter to prescribe particular roles or to tell organizations how to assign them; the purpose is merely to call attention to the need for a clear set of roles and descriptions.

Comparing Software Engineering Workflows

Models can be also be used to compare prescribed process descriptions and observed practice descriptions. This section describes an earlier study performed by Aldo de Moor and myself using a conceptual model to evaluate an existing industrial development process. The study itself was described in (Delugach & de Moor, 2005) and (de Moor & Delugach, 2006); the results are summarized here to illustrate some of the benefits of formal process modeling. This example is different from the previous ones in that the developers were cognizant of the roles involved in their processes and used the above framework to specify two models: (i) their prescribed process from their development guidelines, and (ii) their actual practice instantiated with the names of their actual developers. As we will see, there were some significant differences between them.

Our example was based on a detailed study of a small-sized internal software development group that develops and maintains aerospace software. This particular group is characterized as small (10-20 persons), necessitating multiple roles per

Figure 7. ISO/IEC 12207 problem resolution process



6.8 Problem resolution process

The Problem Resolution Process is a process for analyzing and resolving the problems (including nonconformances), whatever their nature or source, that are discovered during the execution of development, operation, maintenance, or other processes. The objective is to provide a timely, responsible, and documented means to ensure that all discovered problems are analyzed and resolved and trends are recognized.

List of activities. This process consists of the following activities:

1) Process implementation;
2) Problem resolution.

6.8.1 Process implementation. This activity consists of the following task:

6.8.1.1 A problem resolution process shall be established for handling all problems (including nonconformances) detected in the software products and activities. The process shall comply with the following requirements:

a) The process shall be closed-loop, ensuring that: all detected problems are promptly reported and entered into the Problem Resolution Process; action is initiated on them; relevant parties are advised of the existence of the problem as appropriate; causes are identified, analyzed, and, where possible, eliminated; resolution and disposition are achieved; status is tracked and reported; and records of the problems are maintained as stipulated in the contract.

b) The process should contain a scheme for categorizing and prioritizing the problems. Each problem should be classified by the category and priority to facilitate trend analysis and problem resolution.

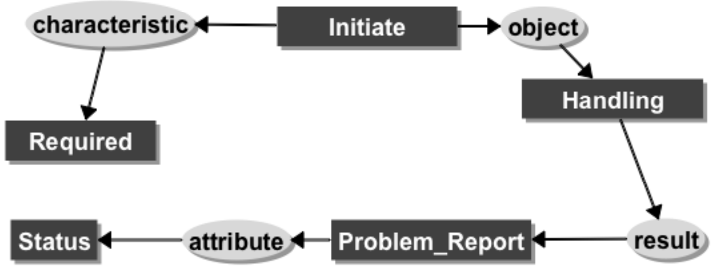
c) Analysis shall be performed to detect trends in the problems reported.

d) Problem resolutions and dispositions shall be evaluated: to evaluate that problems have been resolved, adverse trends have been reversed, and changes have been correctly implemented in the appropriate software products and activities; and to determine whether additional problems have been introduced.

6.8.2 Problem resolution. This activity consists of the following task:

6.8.2.1 When problems (including nonconformances) have been detected in a software product or an activity, a problem report shall be prepared to describe each problem detected. The problem report shall be used as part of the closed-loop process described above: from detection of the problem, through investigation, analysis and resolution of the problem and its cause, and onto trend detection across problems.

Figure 8. Model for standard problem resolution process



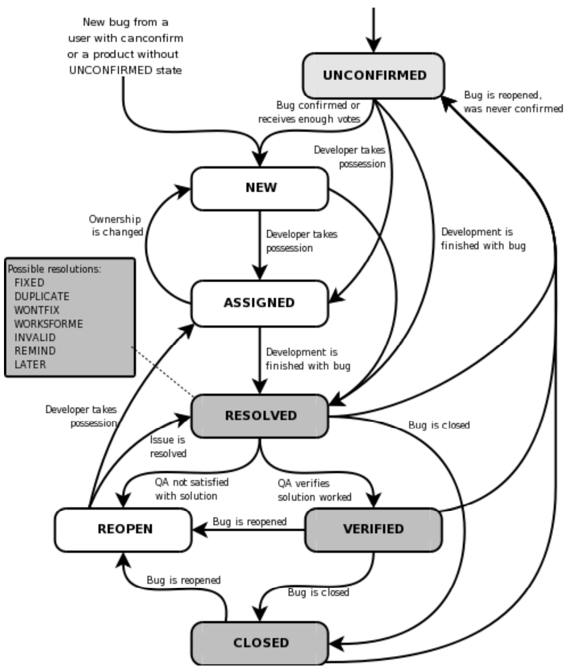
There is an initiate process that is required for handling a problem report which has a status.

person, with little duplication or cross training of roles, some occasional role re-assignment, and (we discovered) implicit accumulating adaptations of the official process in practice.

Software development in this group is project-based; we compared its prescribed process and actual practice models. Space does not permit us to show our complete model of a software process; we focused on one small part: namely, this organization's activity of creating and approving changes to the requirements.

The graphs resulting from this study consisted of a few dozen nodes each. One of them is reproduced here to show the practicality of these models as well as to illustrate the complexity that can arise in even a short process with simple steps. Figure 11 shows the instantiated model of a particular small team's software requirements change process. Each step has an **Initiate**, **Execute** or **Evaluate** activity, with its accompanying role(s), objects and results. There is a new feature of conceptual graphs shown in the model of Figure 11—a dashed line connects

Figure 9. Bugzilla bug tracking process


New bug from a user with canconfirm or a product without UNCONFIRMED state
UNCONFIRMED
Bug confirmed or receives enough votes
Bug is reopened, was never confirmed
Developer takes possession
NEW
Ownership is changed
Developer takes possession
Development is finished with bug
Possible resolutions:
FIXED
DUPLICATE
WONTFIX
WORKSFORME
INVALID
REMIND
LATER
ASSIGNED
Development is finished with bug
Developer takes possession
RESOLVED
Issue is resolved
Bug is closed
QA not satisfied with solution
QA verifies solution worked
REOPEN
Bug is reopened
VERIFIED
Bug is reopened
Bug is closed
CLOSED

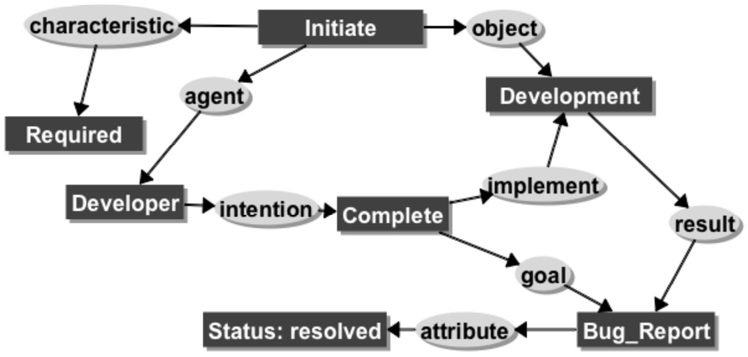
several concepts to each other. This dashed line is called a "co-referent link" or a "line of identity" indicating that the joined concepts refer to the same individual. (This gives the ability to identify individuals without necessarily using exact names.) This is especially important when considering roles, since the lines indicate that the same person serves multiple roles.

Figure 11 shows several steps in sequence, each with a resulting artifact (e.g., **Change_Request**) that is then used by succeeding steps (e.g., relations "**uses**" and "**object**"). For example in the first step, a software engineer "**Jerry**" is permitted to initiate a **MakeRequest** activity and also to execute that same activity, which results in an "**Evaluate**" activity which the software lead "**Terry**" evaluates, and so on. The purpose of this (large) example is twofold: it is a good example of a usable conceptual graph and it shows the large number of relationships in a typical process.

Note that this process model appears more complete than the previous ones in that it does show both roles and their deontic effects. This is an intentional result of the acquisition process by which the model was obtained. A manager was interviewed, with the purpose of explicitly recording roles. For each step, therefore, the manager was asked who did what and what was their deontic role. It is worth noting that the mere asking of these questions would occasionally provoke some thought in the manager about the precision of his process description. Once the prescribed process and actual practice graphs were manually obtained, an automated comparison produced a small list of differences, but those differences were significant from a process-oriented viewpoint.

Figure 12 highlights a key difference between the workflow models. The highlighted portions are

Figure 10. Bugzilla bug tracking process model

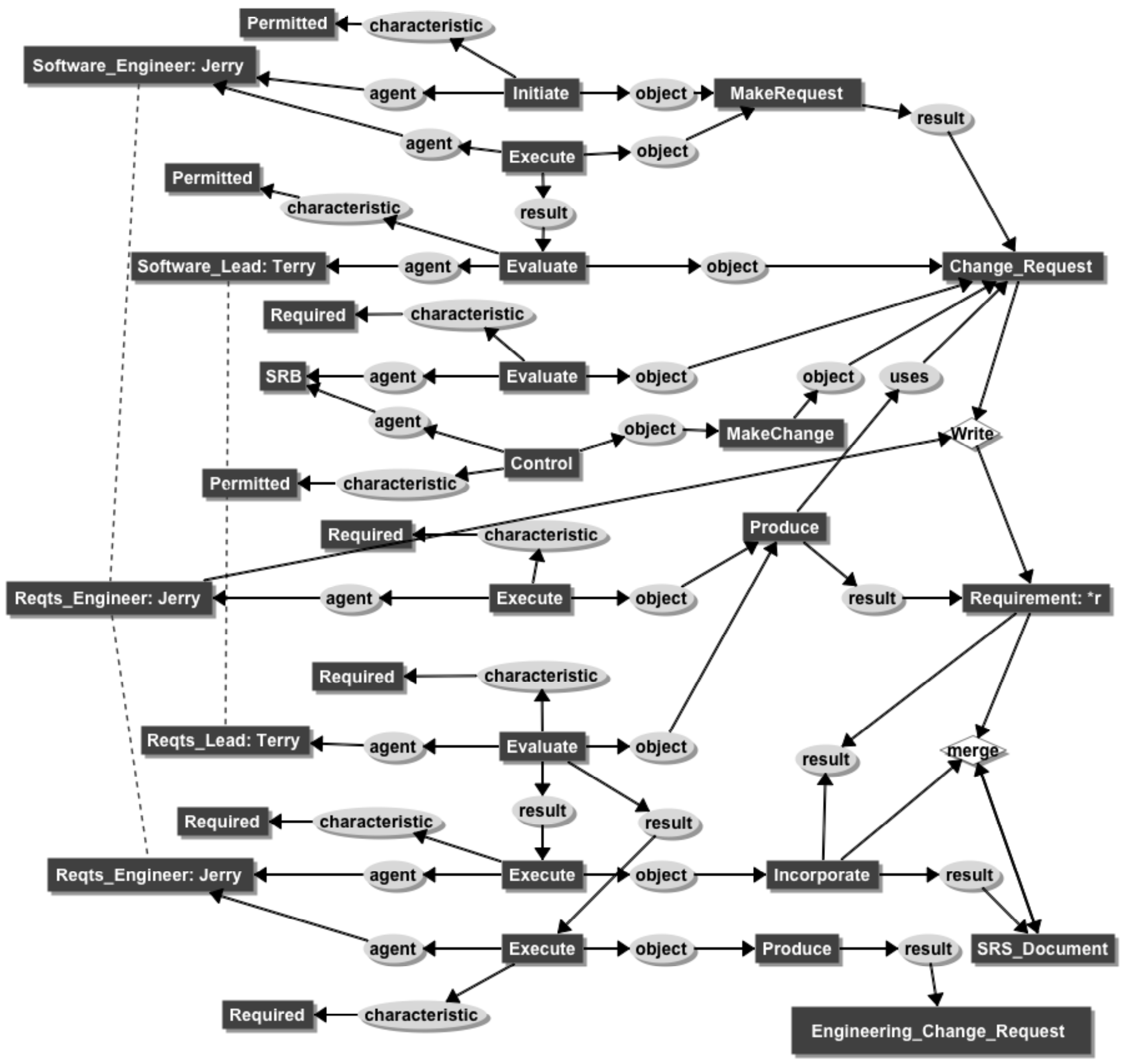


There is a developer who is required to initiate a development that will implement their intention to complete a bug report. The development's result is a bug report with status "resolved".

in white, with the rest of the graph "grayed out". Differences are apparent between how requirements engineers (RE's) are modeled. The prescribed process model on the left—Figure 17(a)—shows a **notEqual** relation between the RE who writes the requirement and the RE who incorporates the requirement into the formal requirements document. This represents the prescribed process constraint that the RE who writes the requirement and the one who incorporates the requirement into the document should be two different people. In the actual practice model on the right—Figure 12(b), however, the RE who produced the requirement (we name them ***r**) is also the same person (referred to as ***s**) who incorporated the requirement into the formal software requirements, a situation that is disallowed due to the **notEqual** relation in the process model.

The point here is that the separation of roles (i.e., the explicit **notEqual** relation) specified in the prescribed process model represents an explicit prohibition, whereas in practice this separation of roles did not occur. This occurred because the

Figure 11. A requirements modification process model instantiated with individuals



same developer ("Jerry") had more multiple roles in the (small) team that happened to coincide for this particular workflow.

Whenever comparisons between models are made, there is always a question about how to proceed: if there is a discrepancy, something is wrong, but what? Which model should be changed? Or should both be changed? Our technique does not prescribe a definitive answer solely from the models; it is up to actual participants such as the software managers and developers to interpret and analyze the comparisons.

Forming Definitions Based on Formal Models

This section of the chapter describes another value in using a model; namely, developing definitions of concepts based on their process relationships. We consider the concept of "status" which often appears in tracking and management artifacts. The particular notion of a bug's status is an interesting one. As one educator reported using the sourceforge tools, "if the phrases describing subtask status are not defined, different student teams often give different meanings to the same phrase. Even worse, sometimes, different members in the same team would interpret the same phrase differently." (Liu, 2005). They identified the need to define status phrases indicating which role and workflow are involved; e.g., the status "Ready for Review" meant ready to be reviewed by the quality assurance (QA) role on the team. A better way to name this would be an explicit "Ready for Review by QA" status.

Another way to envision *status* as a working concept is to approach it from the perspective of its relationships to the concepts in a given workflow: an item's status reflects the process that produced it, not some arbitrary choice from a pull-down menu. So a more accurate and useful definition of status would look something like Figure 13.

This representation shows *status* not as an independent attribute but instead as a dependent attribute—dependent on the process that produced it. This example illustrates another power of conceptual graphs—the graph contains a *context* that allows the modeling of feature clusters. In this case, a definition is described in terms of workflow step features. One can easily envision that, given adequate definitions in a formal model, some characteristics of a process (e.g., "status") would not have to be explicitly stated by participants—they could be derived by observing the current workflow step. This one example is meant to suggest one clear advantage of formal models in their being able to support automatic logical inferences, which is a subject of future research.

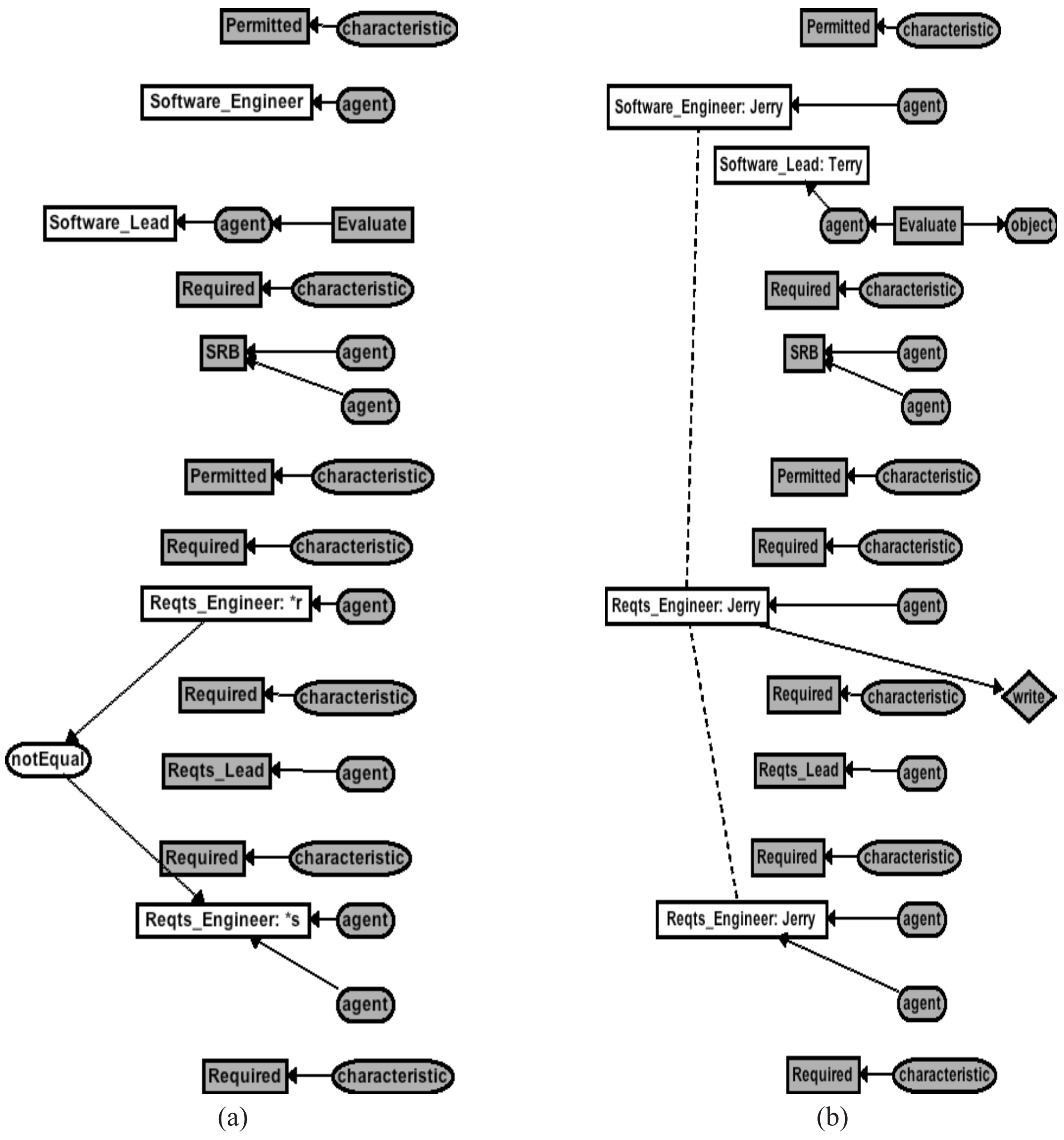
SUMMARY

This chapter was intended as an illustration using formal models of process concepts to describe, analyze, compare and reason about some software development processes. Because most workflow definitions provide only vague (or absent) roles, responsibilities and managerial duties may also be vague. For example, most existing tools do not address the issues of why someone is authorized (or not) to make a change to an item's status, so it is possible for the status of items to be inconsistent with whatever process the software's developers are supposed to follow.

The advantages of using conceptual graphs to represent the workflows are (i) conceptual graphs have the potential to be formally manipulated and compared, and (ii) they provide an easily understood visual description of the process for developers and analysts. In one requirements modification exercise based on this approach, the models' comparison led to a specific potential weakness in the current workflow, toward which a manager was able to focus effort to correct.

For bug-tracking in particular, the subsequent process of how to actually correcting the defects identified during the process, with duties and responsibilities assigned to appropriate roles, is an interesting area to study further, since it involves a superset of the same roles involved in problem tracking. Obviously it will be useful to compare different

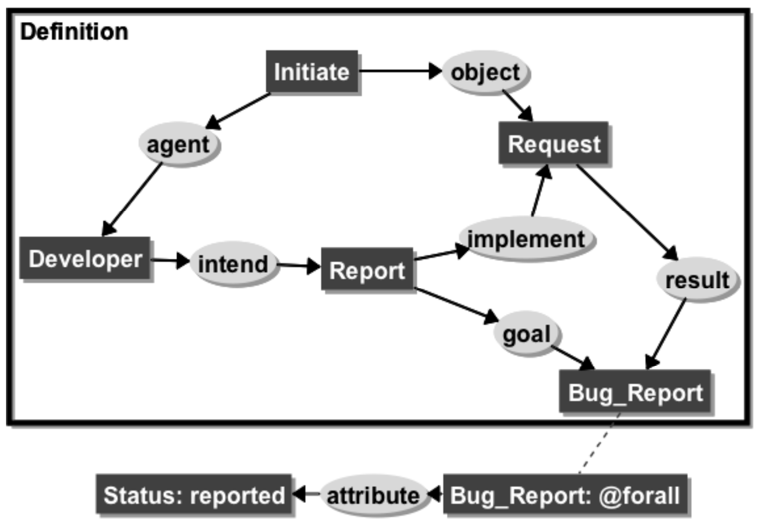
Figure 12. Comparing a prescribed process (a) and an observed process (b) model



organizations' processes to find common features, and (likely) missing features; this is a natural next step. It will also be useful to identify where the processes actually conflict with each other. This last issue becomes quite relevant as companies' products and personnel are merged with other companies' products and personnel.

Using formal models is an important aspect in workflows: models help us conceive, develop, describe, evaluate and compare workflows in system development. This chapter described one technique for representing workflows that is capable of accomplishing all these purposes, with the hope that researchers and practitioners will ultimately benefit.

Figure 13. Status defined as a derived concept



A bug report with status "reported" is defined to be a bug report where a developer has imitated a request for the bug report, and that same developer intends that request to report a bug.

REFERENCES

de Moor, A. (1997). Applying Conceptual Graph Theory to the User-Driven Specification of Network Information Systems. In D. Lukose, H. S. Delugach, M. Keeler, L. Searle & J. F. Sowa (Eds.), *Conceptual Structures: Fulfilling Peirce's Dream*, LNAI Vol. 1257, pp. 536-550): Springer-Verlag.

de Moor, A., & Delugach, H. (2006). Software Process Validation: Comparing Process and Practice Models. *Eleventh Intl. Wkshop on Exploring Modeling Methods in Systems Analysis and Design* (EMMSAD '06). In conjunction with *18th Conf. on Advanced Information Systems Engr.,* Luxembourg.

de Moor, A., & Jeusfeld, M. (2001). Making Workflow Change Acceptable. *Requirements Engineering, 6*(2), 75-96.

Delugach, H., & de Moor, A. (2005). Difference Graphs. In F. Dau, M.-L. Mugnier & G. Stumme (Eds.), *Common Semantics for Sharing Knowledge: Contributions to ICCS 2005* (pp. 41-53). Kassel, Germany: Kassel University Press.

Delugach, H. S. (1992). Specifying Multiple-Viewed Software Requirements With Conceptual Graphs. *Jour. Systems and Software, 19*, 207-224.

Delugach, H. S. (1996). An Approach To Conceptual Feedback In Multiple Viewed Software Requirements Modeling. In *Viewpoints 96: International Workshop on Multiple Perspectives in Software Development* (pp. 242-246). San Francisco.

Delugach, H. S. (2006). Active Knowledge Systems for the Pragmatic Web. In M. Schoop, A. de Moor & J. Dietz (Eds.), *Pragmatic Web: Proc. of the First Intl. Conf. on the Pragmatic Web* (Vol. P-39, pp. 67-80). Stuttgart, Germany: Gesellschaft für Informatik.

Delugach, H. S. (2007). An evaluation of the pragmatics of web-based bug tracking tools. In Buckingham Shum, S., Lind, M. and Weigand, H., (Eds).*Proc. 2nd Intl. Pragmatic Web Conf.*, 22-23 Oct. 2007, Tilburg: NL. ISBN 1-59593-859-1 & 978-1-59593-859-6. ePrint Archive: http://oro.open.ac.uk/9275, pp. 49-55.

Etzkorn, L., & Delugach, H. (2000). Towards a semantic metrics suite for object-oriented design. *Proceedings of the34th International Conference* *on Technology of Object-Oriented Languages and Systems*, (pp. 71-80).

Etzkorn, L. H., Davis, C. G., & Bowen, L. L. (2001). The language of comments in computer software: A sublanguage of English. *Journal of Pragmatics, 33*(11), 1731-1756.

ISO/IEC. (1996). ISO/IEC 12207-0:1996, Standard for Information Technology - Software life cycle processes.

Levinson, W. A., & Rerick, R. A. (2002). *Lean Enterprise: A synergistic approach to minimizing waste*: Amer. Soc. for Quality Management.

Liu, C. (2005). Using Issue Tracking Tools to Facilitate Student Learning of Communication Skills in Software Engineering Courses. *18th Conference on Software Engineering Education and Training (CSEE\&T),* Ottawa, Canada, April.

MCO-MIB-I, Mars Climate Orbiter Mishap Investigation Board, Phase I Report, Nov. 10, 1999, National Aeronautics and Space Administration.

Polovina, S., & Heaton, J. (1992). An Introduction to Conceptual Graphs. *AI Expert*, (pp. 36-43).

Pressman, R. S. (2001). *Software Engineering: A Practitioner's Approach* (5th ed.). New York: McGraw-Hill.

Ripoche, G., & Sansonnet, J. P. (2006). Experiences in Automating the Analysis of Linguistic Interactions for the Study of Distributed Collectives. *Computer Supported Cooperative Work (CSCW), 15*(2-3), 149-183.

Scacchi, W. (2002). Understanding the requirements for developing open source software systems." *IEE Proc.-Softw., 149*(1), 24--39.

Sowa, J. F. (1984). *Conceptual Structures: Information Processing in Mind and Machine*. Reading, MA: Addison-Wesley.

KEY TERMS

Conceptual Graphs: A knowledge modeling approach based on semantic networks and first-order logic, first introduced by Sowa, whereby knowledge is represented by concepts and relations linked together in a bipartite graph.

Deontic Effect: A feature of an activity assigning to it some role's obligations, such as whether the role is required to perform the activity, permitted (but not required) to perform it, or prohibited from performing it.

Formal Model: Any model with well-formed syntax and semantics, such that it is amenable to systematic (usually automatable) processing and analysis subject to logical rules.

Process Model: Any description of a process (not necessarily formal), that shows a series of steps aimed at accomplishing some goal.

Requirements Change Process: A systematic series of steps by which changes to formal software requirements are identified, evaluated, approved and incorporated.

Software Development Process: The overall process of software development, from initial inception through analysis, design, implementation, test and deployment.

Software Issue Tracking: Also called "bug tracking"; a process by which issues (errors, defects, faults, problems) in some software component are identified, evaluated, analyzed, authorized and implemented.

Workflow Model: A process model specifically aimed at representing a development process, as opposed to an algorithm or program.